UNIVERSITY OF PLYMOUTH

PEARL

PHD

A VISUAL DESIGN METHOD AND ITS APPLICATION TO HIGH RELIABILITY HYPERMEDIA SYSTEMS

NEWMAN, ROBERT MALCOLM

Award date:
1998

Awarding institution:
University of Plymouth

Link to publication in PEARL

A Visual Design Method and its Application to High Reliability Hypermedia Systems

R. M. Newman

Ph. D.

1998

A VISUAL DESIGN METHOD AND ITS APPLICATION TO HIGH RELIABILITY HYPERMEDIA SYSTEMS

ROBERT MALCOLM NEWMAN

A thesis submitted in partial fulfilment of the University's requirements for the Degree of Doctor of Philosophy

27 APRIL 1998

Coventry University

Abstract

This work addresses the problem of the production of hypermedia documentation for applications that require high reliability, particularly technical documentation in safety critical industries. One requirement of this application area is for the availability of a task-based organisation, which can guide and monitor such activities as maintenance and repair. In safety critical applications there must be some guarantee that such sequences are correctly presented. Conventional structuring and design methods for hypermedia systems do not allow such guarantees to be made. A formal design method that is based on a process algebra is proposed as a solution to this problem. Design methods of this kind need to be accessible to information designers. This is achieved by use of a technique already familiar to them: the storyboard. By development of a storyboard notation that is syntactically equivalent to a process algebra a bridge is made between information design and computer science, allowing formal analysis and refinement of the specification drafted by information designers. Process algebras produce imperative structures that do not map easily into the declarative formats used for some hypermedia systems, but can be translated into concurrent programs. This translation process, into a language developed by the author, called ClassiC, is illustrated and the properties that make ClassiC a suitable implementation target discussed. Other possible implementation targets are evaluated, and a comparative illustration given of translation into another likely target, Java.

Contents

Chapter 1: Introduction 6

Chapter 2: Hypermedia documentation for high reliability applications 11

2.1 Introduction 11

2.2 The move towards hypermedia technical documentation 11

2.3 User's requirements 16

2.4 Safety Critical Industries 19

2.5 Design Principles 21

2.6 Design methods for hypermedia systems 23

2.7 The need for a methodology 26

Chapter 3: Current work in the design of hypermedia systems 29

3.1 Areas of work in hypermedia systems 29

3.2 Content Authoring 30

3.3 Content Presentation 35

3.4 Content Structure 37

3.5 Database Organisation 42

3.6 Navigation 46

3.7 Program based organisation 47

3.8 Relationships with this work 47

Chapter 4: A methodology for design of large hypermedia systems 51
4.1 Introduction 51
4.2 High reliability large Hypermedia databases 52
4.3 Existing design methods 55
4.4 Multimedia as document 56
4.5 Multimedia as game 57
4.6 Multimedia as movie 58
4.7 Multimedia as database 59
4.8 Multimedia as program 60
4.9 Multimedia as hypermedia 61
4.10 Design choices for the hypermedia designer 62
4.11 Methods and notations 65
4.12 Selecting a process algebra 71
4.13 Dealing with size and complexity 74
4.14 Framing safety conditions 76
4.15 Applying process algebras to hypermedia systems 77
4.16 A graphical notation 82
4.17 Software tools 85
4.18 Analysing and proving designs 87
4.19 Structures produced using process algebras 90
4.20 How the method is used 90
4.21 Separation of structure and content 92
4.22 An example 93
4.23 Comparison with Eventor 99
4.24 Conclusions 103

Chapter 5: Implementing high reliability multimedia systems ... 105
5.1 Introduction ... 105
5.2 The form of the specification ... 107
5.3 Introduction to COOLs ... 112
Chapter 6: Translating CCS specifications to COOLs ... 115
6.2 The ClassiC language ... 118
6.3 Translating CCS specifications into ClassiC ... 153
6.4 Conclusion ... 160
Chapter 7: Java, JavaScript and HyTime ... 162
7.1 HyTime ... 163
7.2 Java ... 163
7.3 JavaScript ... 169
7.4 Comparing Java, JavaScript and ClassiC ... 171
7.5 Translation Rules to Java ... 172
7.6 Including text, sound, animation and models ... 178
7.7 Conclusion ... 179
Chapter 8: Conclusion ... 181
8.1 Results ... 181
8.2 Issues ... 182
8.3 Future work ... 183

Chapter 1: Introduction

This thesis is the synthesis of two different of two different areas of work which together provide an original solution to one of the major problems to be faced in the building of hypermedia technical documentation systems for safety critical applications.

For some time the author has had both a research and a teaching interest in real-time systems, particularly concurrent object oriented programming systems and event based specification methods. This has resulted in a body of work including the development of a concurrent object-oriented programming language and the exploration of the consequences of particular aspects of the language design, in particular he mechanisms adopted for process instantiation, inter process communication and the provision of non-determinacy.

More recently the author has been engaged in a European Union funded research project investigating the requirements for the production of large multimedia technical documentation systems. One of the outcomes of this work has been the realisation that a multimedia information system can be seen as a variety of concurrent program. It has also become clear that some documentation systems may truly be as "safety-critical" as the real

time computer systems in the products that they document. One consequence of this is that there is a need for design methodologies that bring increased rigour to the design process of such systems. To this end, the previous work on the formal semantics of concurrent programming has been applied to the application are of multimedia systems design. The outcome is a method of multimedia systems design based on the formal methods used in concurrent systems design. In order to make this method acceptable to information designers a semi-graphical specification language has been devised with the general appearance of a storyboard, a design planning concept familiar to many multimedia designers. One of the attractive features of this notation is that it can form a "bridge" between the non-formal, practice based world of the information designers who are likely to be responsible for the design of such systems and the software engineers who will be responsible for their implementation and verification.

This thesis is composed of eight chapters, of which this is the first. Chapter Two discusses the application area of multimedia for technical documentation, drawing particularly on the investigations undertaken in the project mentioned above. It explains that there is a widespread need in the engineering industries to be able to produce high-quality multimedia documentation systems, for both economic and practical reasons. The important issues of the field are explored. Central to those issues is the need for methods of authorship that can guarantee high quality, in terms of correctness, systems and maintain a high or increased level of authoring productivity. The likely user requirements, in terms of constituent media,

applications and usage of such systems are discussed. In particular, it is proposed that in some applications the technical documentation is "safety-critical", in that errors in the documentation system can cause systems failures due to maintenance errors. For this reason it is suggested that rigorous methods for design of hypermedia systems are required. Methods currently used are discussed in this light.

Chapter Three surveys the current state of the art in the design and maintenance of large multimedia systems. Most of this work has been in the domain of construction of multimedia databases and the query mechanisms to go with them. It is argued that a database model is not particularly suitable for maintenance documentation systems, for which a task-directed, procedural design is more suitable. Little work has been done on methods for producing large-scale systems according to such a model. What work has been done has tended to concentrate on interactive authoring techniques or development of scripting languages.

Chapter Four introduces the new method for designing these systems. The method is based on process algebras. The background of these is explained and a rationale given for their selection as a suitable starting point for the new method, along with the reasoning behind the selection of the process algebra, CCS, which underpins the new method. The method uses a semi-graphical notation which combines elements of the traditional storyboard, used for planning films and more recently multimedia presentations, while at the same time including the symbolic content of CCS. This allows designs produced in this notation to be

translated to CCS to allow analysis and verification to occur using established methods. This process is explained and a worked example given.

Chapter Five explains how designs produced using the method may be refined into working multimedia systems. The structures produced by the method are best implemented using a structured scripting or programming language. One class of language, concurrent object-oriented languages, or COOLs, has characteristics that make the languages that belong to it suitable targets. This class of programming language is introduced in this chapter.

The COOL that the author has developed, ClassiC, is introduced in Chapter Six and an illustration of the translation process is given. The design features of ClassiC, which render it a particularly simple translation target for this type of system are discussed.

Unfortunately, ClassiC is unlikely to be available to implementers of hypermedia systems, and they are likely to have to use established hypermedia languages such as Java, JavaScript or HyTime. ClassiC has many similarities to Java, and, since JavaScript is derived from Java, to JavaScript as well. The three languages are compared in Chapter Seven and the applicability of the work on ClassiC to Java is established and it is shown how translation of CCS specifications to Java may be achieved. These translation rules are compared with those to ClassiC.

Chapter Eight is the conclusion of the thesis. It draws together the pieces of work and proposes how the various methods introduced in the

work might be combined together with appropriate tools to develop a complete methodology for development of large multimedia systems for technical documentation.

Chapter 2: Hypermedia documentation for high reliability applications

2.1 Introduction

This chapter discusses the requirements for authoring methodologies for large multimedia systems to be used for technical documentation. It argues that some fundamental issues concerned in the production of these systems are often overlooked. These issues include those of the verifiable correctness of such systems, both in terms of their content and other issues such as sequence of presentation. It is argued that the addressing of these issues is essential to the development of technical documentation systems that are of sufficient quality to be used for safety critical applications such as the transport industry. The requirements that viable design methodologies for these applications must address are discussed, providing a research agenda for the field of design methods for technical documentation multimedia systems.

2.2 The move towards hypermedia technical documentation

Maintenance documentation is an application domain that seems tailor made for multimedia systems. The ability to demonstrate

maintenance procedures using advanced graphics and animation, to offer on-line diagnostic support and the promise of a replacement for the inconvenience of paper based media in such a situation suggest that this technology will provide greatly enhanced documentation support for maintenance organisations.

There are many different motivations for the adoption of hypermedia documentation. Some of these have been researched in the studies undertaken by the Online Multimedia Information for Maintenance and Operation (OMIMO) project [Newman et. al. 1997]. This was a feasibility project funded under the Telematics Applications Programme of the European Union. The consortium members of the project were the Visual and Information Design Research Centre at Coventry University, Rolls-Royce PLC, VTT - the Finnish national research agency, Etnoteam S.p.A., and Caplan Systems and Research. As a part of the project VTT undertook a survey, based on their previous work in this area [VTT Automation, 1996] of nine selected Finnish companies, particularly those producing capital goods. The findings of this survey were that most companies considered that multimedia documentation would be important in the future, but that there was a decided reluctance to invest, partially due to the perceived risks of the development process. One large company, Kone Elevators, took a much more positive view. The reasons that they gave for the importance of hypermedia documentation were as follows.

Firstly there was the scale of the technical documentation operation in their organisation. Documentation consumed 20-40% of company costs.

With this scale of spend, any reduction of documentation costs would have a dramatic effect on the business. The ways in which electronic documentation might contribute efficiency gains are discussed below.

Secondly, there are distribution difficulties. The company operates in countries all over the world, maintaining more than 450 000 elevators, almost all maintenance is on site and there is large variation between individual installations – only 50% of elevators are considered to be volume "products" and nearly a quarter were manufactured by other companies. The maintenance distribution of such a diverse collection of maintenance documentation presents a major problem. The use of information technology, together with networked communication, is seen to offer one route to the solution of this problem.

The third reason given was that use of advanced technology for support documentation was seen to give a possible marketing advantage. It was felt that its support operation could project a very efficient and forward-looking image by making use of such systems.

Rolls-Royce documented their experiences and requirements in the project's deliverables. The company is rather more advanced in adoption of hypermedia documentation, having already produced one system, DRUID, and is maintaining a major company initiative to continue development in this field. They state their reasons for moving towards hypermedia documentation as follows. [Newman et. al. 1997]

> It is widely recognised that the use of physical documentation incurs heavy costs, constrains the effectiveness of communication and can be time consuming with respect to finding the required information.

> The application of multi-media on-line technology has the potential to better meet the information needs of the operator and maintainer.

At the same time, their previous attempts to solve these problems have not been universally successful.

> Non-digital information such as paper manuals do offer some advantages over digital methods, e.g. portability and cost. Future digital systems must ensure that they offer superior performance with respect to all criteria.

There is widespread activity in the field within the aerospace industry as a whole. Systems have been developed by British Airways (DISC system, described in [Jones, 1991], Luftansa's BISAM [Orlowski, 1995] and the widely publicised example of the Boeing On-Line Documentation system ,BOLD, developed alongside the 777, described in [SITA , 1996] and many other places. There are similar activities underway at Aerospatiale, Airbus and Alenia Aerospace and, in the motor industry, Rover and BMW. Public information and descriptions in the literature in such systems tends to be scarce for reasons of commercial confidentiality and the traditions of technical publishing departments who are generally responsible for such work.

The US Department of Defense has for some years been promoting hypermedia documentation systems development as part of its Continuous Acquisition and Life-Cycle Support (CALS) Initiative [Department of Defense, 1994]. The aim of CALS is to migrate from paper intensive documentation systems to highly automated acquisition and support processes. Potential benefits are stated to be:

Improved information quality for acquisition, management, re-procurement and maintenance.

Reduced acquisition and support costs through elimination of duplicative, manual and error-prone processes.

Reduced space, weight and storage requirement for digital media (in comparison with paper media)

Increased responsiveness to industrial base through development of integrated design and manufacturing capabilities.

One central part of the CALS strategy is the Contractor Integrated Technical Information Service (CITIS) [Department of Defense, 1993], in which a customer (the Department of Defense) will have direct access to contractors documentation databases. Within the initiative are a number of standards defining such things as interaction style [Mil-M-87268 (GCSFUI) 1992] and database services [IETM 1992]

There has also been a substantial amount of work done and reported to prototype or demonstrate such systems by research institutions, including the work by Fischer [Fischer, 1997] at Coventry University, Farrington [Farrington, 1994] and the Engineering Research and Development Centre (EDRC) at Hertfordshire University [Wu et. al, 1997], related to the aerospace industry, and by Alty and Bergan [Alty, Bergan 1993, 1995] at Loughborough University for the nuclear process industry.

The reasons for making the (at that time, prospective) switch to electronic documentation had been rehearsed in 1988 by Ventura [Ventura, 1988] and re-iterated by Horton in 1993 [Horton, 1993]. This argument relates mostly to the volume and complexity of modern technical documentation. An example given is the comparison between the Piper

Cub aeroplane's maintenance documentation during the Second World War, which consisted of only 20 pages, a mid 60's F 101 B fighter, which required 25 000 pages and the current F-15 fighter's technical information, which needs over 1 million documentation pages to be fully operational. Electronic documentation, it is proposed, offers the means to store the documentation compactly and to retrieve it easily.

2.3 User's requirements

The scope and ambition of these systems varies considerably. Some, such as the Rolls-Royce DRUID system, are essentially replicas of the paper based documentation. The aim has been to utilise the advantages of digital storage systems to reduce the bulk and publication cost of paper systems by distributing using a digital medium such as a CD. In the DRUID system the opportunity has been taken to enhance the page based system by addition of embedded links in the pages, allowing direct access to references in the field. It is difficult to determine whether this enhancement, by itself, makes a qualitative improvement to the documentation, since the system itself was little used. (The major reason for this was that it was not portable, and therefore not available on-site).

The OMIMO project spent considerable effort researching user's requirements for hypermedia technical documentation. To a large extent this activity was constrained by the lack of awareness of many companies of the nature of the technology, and a corresponding inability to frame requirements. Thus the requirements were largely drawn up by Kone

Elevators, working with VTT, and Rolls-Royce. In addition service personnel from British Airways, NAYAK Aircraft Service and Lufthansa were interviewed. These results are set out in [Fischer et al, 1996]. Those that are summarised here are selected to include only requirements related to hypermedia systems (the project envisaged a complete service support structure, which integrated documentation with communications, history logging, and interlinked with stock, catalogue and other management systems). These requirements are numbered for clarity of future reference. There is no intended priority.

1. Simplicity of navigation was an often-cited requirement. Often this requirement was derived from direct experience of using hypermedia documentation. The experience of becoming "lost in hyperspace" [Edwards, Hardman 1989] was felt to be a major drawback to acceptability and usability of hypermedia subsystems. Linked to this requirement, fragmented, multi-windowed presentation of information, as is presented by browsers following a link organised hypermedia system, was considered undesirable. An integrated, planned presentation style was considered preferable.
2. Robust reliable and timely delivery of information at the point of use, generally users are not interested in printing off data for use in the field. Some experience with existing systems has suggested that they are too slow to be usable. Boeing has suggested 15 seconds as the maximum acceptable response

time for online information. The ideal was felt to be an interactive system portable enough for service engineers to use in the field, and to interact with while performing the maintenance tasks. Ideally the system should guide the engineer through the task, and document completion of it.

3. Interactive support is considered important. This is supported by Fischer's evaluation of his prototype [Fischer, 1997], where his dynamic and animated documentation system was significantly more effective in communicating maintenance task information than the non-interactive alternatives. Rolls-Royce commented that animated systems diagrams had proved to be effective.

4. Rolls-Royce felt that availability of 3-D models in the documentation was important. They commented that:

   > 3-D geometry viewing of products with simple and responsive interface that allows natural walk around and inspection of product [is a requirement]. At Rolls-Royce fitters found this far superior to illustration in many situations.

   This requirement was directly contradicted by most other companies studied, who felt that 3-D models were an unnecessary complication (although there is no evidence that this view is actually based on experience of their use).

5. It was felt that the documentation had to be differently organised for different tasks. Current systems (including the paper based ones) have rigid organisation that doesn't support any task

particularly well. Unstructured systems suffer from the navigation problems noted above.

6. The two most important roles identified for maintenance documentation was in the field support for service engineers and for training of service engineers.

7. Integration of the various supporting systems within the documentation system is required. An example given is of a person performing a repair, who needs access to part details, tools, consumables, facilities and workflow instructions. However it is stated that this integration must be achieved without the "fragmentation" typical of hyper-structures.

8. The hypermedia documentation system must be complete, that is it must provide all the documentation resources necessary, without the need to keep backup paper or microfiche systems. Neither the manufacturers nor the users have any desire to maintain two parallel documentation systems.

With the exception of the disagreement noted on the use of 3-D models there was surprising unanimity as to the failures of current documentation systems and the type of properties needed by hypermedia systems that will replace them.

2.4 Safety Critical Industries

Although there was no particular prioritisation intended in the list of requirements stated above, some are clearly very important indeed. It is

worth noting that all the industries examined are in some sense "safety critical", that is the result of systems failure could very possibly be loss of life. As a result of this many of these industries exist in a regulatory framework which controls the way they design and operate their products. The Aerospace industry is a good example of this. Standards, generally set out by the national aviation authorities, exist that dictate design practice and maintenance procedures. Such standards extend to software design, including specification and implementation methods. As concern for safety increases, and the occasional incident of an accident caused by software failure gains widespread publicity, there is increasing pressure for the adoption of formal methods of specification and verification of systems software. For instance, SNCF, the French Railway Company, now insists that all embedded software system used by the railway are specified and verified using the B method [Bieber, April 1996, December 1996].

There are arguments that hypermedia technical documentation should be similarly regarded as safety critical software. If we examine the requirements given above, 3 and 5 suggest that direct interactive support should be given for tasks such as maintenance, which suggests that the system will guide the engineer through the maintenance processes. Requirement 7 suggests that the engineer should be able to access additional information related to the task in hand, but without the "fragmentation" caused by multiple concurrent contexts. What this suggests is a system that leads the engineer along the maintenance path, but allows diversions to explore such things as spares availability. If, after the

diversion, the user was led to the wrong part in the sequence, so that, maybe, a vital step in the maintenance sequence was missed, then the results could quite easily be life threatening. Thus one is led to the conclusion that maintenance support systems are also safety critical software, and likely at some point to be subject to the same rigour demanded for embedded systems software in the same industries. Such design rigour demands a rigorous design method to achieve it. Such a method must be developed for hypermedia systems design. It must be usable by the people responsible for authoring documentation systems who are often information designers, not software engineers.

2.5 Design Principles

Rubens and Krull [Rubens, Krull 1988] have classified many types of on-line information, most of which can be argued to be present in some form in technical documentation. For the purpose of this work we will focus of those most relevant ones for in the field maintenance systems, which are support for interactive tasks and tutorial and canned demonstration. For both of these the most commonly used authoring styles, and those that the users support most strongly, are task based, narrative styled documentation, that leads the user, step by step, through the maintenance task in hand. The need for this type of organisation has been argued before, specifically in the context of aircraft maintenance by Taylor [Taylor, 1990] in which an organisation is described where the engine management schedule is embodied in documents or work flow software which triggers different types of operational and maintenance.

In order to meet requirement 5, above, in the context of maintenance support it is likely that the essential structure of the system should be task oriented. The idea of designing systems using a task-oriented approach has gained considerable support, particularly in the field of user interface design. Here the idea is use *task analysis* to analyse the tasks that the system user is carrying out and design the system to match the requirements of that task. One commonly used task analysis method is Task Knowledge Structures (TKS) [Johnson et al, 1988]. Sutcliffe and Faraday [Sutcliffe, Faraday, 1994] describe a method for the selection of suitable media and interaction dialogues for multi-media systems, based on task analysis. Benyon [Benyon, 1992] discusses the use of task analysis for the design of interactive computer based systems, and the relationship to systems analysis. He notes the weaknesses, mainly due to loss of system structure, that may be introduced if the task-based design is not informed by principles of systems analysis. User interface design using task analysis as the starting point is described by Copas and Edmonds [Copas, Edmonds, 1994]. They describe executable task analysis for production of user interfaces and discuss the issues of integration. Casner [Casner, 1991] describes a system for automated design of "graphic presentations" which is based on an analysis of the task which the graphic is designed to support. This is developed by substituting logical inferences with perceptual inferences in a way that is claimed to be provably equivalent. It is claimed that such design demonstrably reduces users' task performance time. Faraday and Sutcliffe apply task analysis to the issue of multimedia interface design, presenting a method based on the technique. The

AMTOSS system referred to above is also based on task oriented design principles.[Farringdon, 1994].

In summary, there is a considerable body of work supporting the application of task-based design to interactive and hypermedia systems. The task analysis methods, such as TKS, described above are used to model the nature of complex tasks, the structure of which may not be apparent by simple inspection. In the field of maintenance documentation the tasks are usually well documented and explicit, so in many cases the job of task analysis will have already been done by the designers of the maintenance procedures.

2.6 Design methods for hypermedia systems.

Faraday and Sutcliffe [Sutcliffe, Fraday, 1994] counterpose their design method to the "intuitive" design process commonly used for multimedia systems. This characterisation of general practice is not entirely fair, since there is a design method generally taught (see for instance the course notes of Hogg at Sunderland University [Hogg, 95]) and used and it is task oriented. The task analysis takes the form of a *storyboard*, a comic strip like sequence of illustrations, in which the illustrations document key events in the task and the sequence of frames indicates the ordering of those events. This method is described in a number of textbooks, as for instance [Bunzel, Morris, 1992] [Heller, Heller, 1996] [Murie, 1994] [Schwier, Misanchuk, 1993] [Bergman, Moore, 1990] and is described in some case studies [Fallenstein-Hellman, James, 1995]. Some of these

texts [Schwier, Misanchuk, 1993] [Bergamn, Moore, 1990] suggest extended storyboards with annotation to indicate interaction but these suggestions, and the use of storyboards themselves, are not well underpinned by research reported in the literature. The storyboard technique comes from movie making practice, and is a natural method to be adopted by those who view multimedia as a kind of interactive movie. Obviously the simple sequence of a storyboard is not sufficient to represent the interactive nature of hypermedia, whereby the user can interact with the system and change the sequence of images or other media presented. This is often handled by embedding storyboard sequences in a *flowchart*, which provides a simple programming structure which is reflected in the operators of the scripting languages commonly used to program such systems. As stated above, there is little mention of this method in the literature outside multimedia authoring textbooks, one of the fullest treatments being given in [Bunzel, Morris 1992], but it does appear to be general practice within the multimedia content authoring industry. Certainly the majority of authoring tools support such a method either explicitly or implicitly. There are several companies that provide a direct mail (or e-mail) order service[1], translating storyboard ideas into multimedia presentations. It is also the method generally employed for the design of computer games, where the implementation vehicle is more likely to be a programming language, and now there are general purpose programming tools appearing which give direct support to "design by storyboard".

---

[1] See, for instance, the service offered by E-media at http://www.e-

If a widely accepted design method for hypermedia systems exists, why can't it be applied to the production of technical documentation? There are a number of reasons

Firstly, the control structures (flowcharts) belong in a bygone age of programming. They reflect directly the control structures of primitive, first and second generation programming languages such as assembly code, FORTRAN and BASIC. These languages have been superseded for any software with any pretensions to reliability because the unstructured control flows that they allow almost inevitably lead to programs that cannot be analysed, verified or even debugged. This was first noted by Dijkstra [Dijkstra, 1968], and although the view was controversial at the time in the world of software systems construction such languages, termed *unstructured programming languages*, are now practically unused.

Secondly, the storyboard/flowchart has no formal semantic model associated with it. This means that it is unsuitable as a starting point for any rigorous development or analysis process.

Thirdly, the method relates to "programming in the small". Methods for designing large software systems must handle the interfaces between the different people working to develop the system. This involves such concepts as modularity, the clean separation of the system into different components, unambiguous definition of the interface and function of different modules and the hiding of the internal workings of modules, lest

---

media.com/german/storybrd-g.html

other parts of the system unintentionally affect them. The storyboard/flowchart has none of these characteristics.

2.7 The need for a methodology.

The widely used methods described above do not form the basis for a methodology for hypermedia design, in the sense of the word used by software engineers. The design methods or theories described by Sutcliffe and Faraday or by Fischer are not methodologies in the software engineering sense. They are concerned with means of producing designs or specifications for systems that best match the goals or psychological or cognitive characteristics of the user community at which they are aimed. This is largely to do with the ergonomics of interaction of the system, and are aimed at producing an optimum design, or specification for the system for the conditions in which it will be used.

The overloading of the word "design" can cause much confusion. For the software engineer, the process design starts when the specification is complete. For the designer, that is when it stops! Software design methodologies are to do with ensuring that the finished product actually performs in the way that was originally specified. That achieving this goal is not trivial is attested to by the wealth of software engineering methodologies that have appeared in the last two decades.

In 1993 Alty [Alty, 1993] observed that the emergent technology of multimedia was being driven by the increasing power and availability of the enabling technology, but that there was not the methodological

development to match the technological development. The situation has not noticeably improved since then. There is an assumption that once the system has been designed (in the designer's sense of the word) that the technology will deliver a faithful realisation of that design. This may be true for simple systems, as it is for simple programs. However, the analogy of program development (of which multimedia development is, after all, a part) suggests that as the system grows to a certain complexity we can have no such assurance. Particularly for safety critical applications, which, we have argued, include technical documentation, there must be a software design methodology, based on a sound theory, to ensure this.

Much of the methodological work in multimedia and hypermedia, surveyed in the following chapter, has concentrated on a different, but related, problem. This work acknowledges the problem of assembling huge, heterogeneous collections of information in different media and producing a system that handles all the media correctly. This is undoubtedly a major issue that must be, and is being, addressed, but is ultimately rooted in a view that sees hypermedia systems as being collections that are assembled, rather than an integrated piece of documentation that is designed. The design of hypermedia systems is, as noted above, an essentially similar problem to the design of large software systems. This being the case, it might be expected that methods established in that field would be simply transferable.

Jeffcoate, in her survey of multimedia technology [Jeffcoate,1995] notes that:

> The existing process of systems analysis is neither appropriate nor sufficient for the development of multimedia systems. This is because building a multimedia application will involve parallel streams of activity to create the content and develop the computer program that will create it.

The nub of the problem with designing a methodology for hypermedia is that it must be usable by all the people involved in the production of the systems. Current systems analysis and development methodologies are really only usable by those with a substantial background in discrete mathematics and formal logic, a group that does not even include all people trained in computer science, yet alone the information designers, illustrators, directors and technical authors who will need to subscribe to a methodology for hypermedia design. If hypermedia is to be used for the production of the only technical documentation system for highly complex, safety critical products, in systems comprising millions of pages of information, then it is essential that such a methodology be developed. Such a methodology will be based around a specification method that can form a bridge between the "creative" world of the information designer and the "formal" world of the software engineer.

Chapter 3: Current work in the design of hypermedia systems

3.1 Areas of work in hypermedia systems

Hypemedia is a technology that has developed in an evolutionary manner, as a hybrid of hypertext and multimedia. Hypertext structures are dynamic documents in which each node or page contains links to other pages, leading to a non-linear structure, as opposed to the linear structure of traditional books, paper documents or word processors files or other computer structures based on them. Multimedia has come to refer to computer systems that integrate together representations of objects in media other than text with textual information. Such media may include images of several types, animations and movies, sound and interactive objects including navigable 3-D models – although the latter are generally held to be in the domain of "virtual reality" as opposed to hypermedia. The combination of hypertext and multimedia gives hypermedia. A definition of this medium has been given as [Halasz 1988]

> the style of building systems for the creation, manipulation, presentation and representation of information in which: the information is stored in a collection of multi-media nodes; the nodes are explicitly or implicitly organised into one or more structures

> (commonly a network of nodes connected by links); users can access information by navigating over or through the available information structures.

This definition provides a starting point for the classification of work in hypermedia systems. Creation and manipulation of the contents of hypermedia documents have together provided a large number of topics for investigation which might broadly be grouped together under the heading *Content Authoring.* Similarly investigations of presentation and representation will be considered under the heading *Content Presentation.* The next headings are to do with the explicit (link based) or implicit (database based or program based) organisation of the system. These are considered under the headings *Content Structure*, *Database Organisation* and *Program Based Organisation* respectively. Finally there is the issue of *Navigation.*

3.2 Content Authoring

Any feasibility study for implementation of a major documentation system using hypermedia will indicate that authoring time and cost is a major obstacle. There are two independent concerns, both of which need substantial improvements in productivity if hypermedia technical documentation is to be feasible when assessed against economic and time metrics. The first of these is content generation, ensuring that the information contained in the hyperbase is generated effectively and easily. The second is structure authoring, the provision of means for the imposition by the author of the required navigation structure on the hyperbase, within the underlying structural architecture of the hypermedia system. Of course,

often the two can become mixed together, especially when using embedded link systems such as HTML.

Much of the work on making content generation more productive has concentrated on the concept of "data mining". The goal of data mining has been defined [Kuntz1996] to be

> ...to enable database users to get more and better information out of the data that they possess and to perceive regularities or kinds of coherence that would otherwise go unnoticed. Ultimately this better understanding of the data overall enables conclusions to be drawn that are impossible to discover from all the data records taken individually.

Briefly, the aim is to use automatic or semi-automatic search tools to hunt in a database or group of databases and find information that may be of value in a hypermedia system. One mechanism has been defined by Kuntz as *scavenging*. This is a hybrid process whereby the user browses the database, in the sense of interactively following a set of data values, rather than following links, and the system makes co-operative queries of the database based on 'learning' of the user's requirements. Other work on data mining is described in [Brachman, Anand 1994] [Faloutson, Lin 1994]. Loosely related to the field of data mining is content based retrieval. In the context of content generation, content based retrieval is used to locate suitable data for inclusion in the hyperbase. It can also be used as a navigation method in its own right and is dealt with later in that context. Various approaches to content based retrieval, mostly applied to image databases, are described in [Chiueh,1994][Faloutsos et. al. 1994][Mehrotra, Gary 1995].

Another approach to tackling the issue of sourcing data is that of *open hypermedia systems*. Such systems, which are open at the data format level aim to aid the availability of data by being able to build hypermedia from a variety of data sources. One approach is by the building of systems that can use data in a variety of different forms in an open ended way. Such an approach is exemplified by the Microcosm system designed at Southampton University [Fountain et. al. 1990]. This system is well described in [Goose 1997], as well as other sources. The other approach to data format openness is openness by translation. The key to this is a common format that is a suitable target for all of the source data formats envisaged, and is open to enhancement as new formats appear. Such common formats are obvious targets for standardisation, with existing, and proposed formats including HyTime (Hypermedia/Time Based Structuring Language)[International Standards Organisation 1992] and MHEG (Multimedia and Hypermedia information coding Experts Group) [Bertrand, Colaitis, Leger 1992]. These two views of openness spring from fundamentally different views of the nature of a hypermedia system. The former approach is based on a view that sees hypermedia systems as heterogeneous collections of data from diverse sources, while the latter sees it as an integrated, and authored, system containing objects with heterogeneous behaviours. One might expect the former approach to suit distributed, decentralised, systems such as the World Wide Web or its descendants while the latter would be better suited to complete hypermedia systems for a limited user community, conceived and designed as a whole but possibly using data from a variety of sources.

The explicit authoring of the structure of such a hypermedia system is another area of research. Four different classes of authoring tools have been identified. [Hardman, Bulterman, 1995]

- *Structure-based* authoring systems support the explicit representation of the structure of a presentation. This gives the advantage of being able to group items together in terms of "mini-presentations" which can be manipulated as a whole. Another advantage can be given by deriving the timing relations in the presentation from the structure, so that alterations in durations of objects are propagated through the presentation by the system.
- *Timelines* show the constituent media items placed along a time axis, possibly on different tracks. These are useful for giving an overview of which objects are placed on the screen when.
- A *flowchart* gives the author a visual representation of the commands describing a presentation. While systems using this approach are deemed simpler to use, they tend to become unwieldy for large presentations.
- A *script-based* system provides the author with a language where positions and timings of individual objects can be specified. Although scripting languages provide a flexible authoring interface, they have the disadvantage of becoming unmanageable in large presentations. Structures such as scene boundaries or timing relations between media items are difficult to recognise in the script.

In the cited work, the characteristics of a number of authoring systems are discussed in detail, both commercial products and those cited in the literature. These include CMIFed, Athena Muse, MET++ and Mbuild, which are structure authoring systems; Director, the Integrator and MAEstro which are timeline based authoring systems; Authorware, IconAuthor,and Eventor which are flowchart based systems and Videobook and Harmony which are script based systems.

Given the perceived weaknesses of timelines, flowcharts and script-based approaches identified by Hardman and Bulterman, most work

addressing authoring systems for large hypermedia systems has concentrated on a structured approach.

Yu and Xiang Describe a system in which the temporal and spatial structure is abstracted away from conventional geometric page layout [Yu,Xiang 1995]. Other work has investigated integrated structure editors [Hardman, Rossum, Bulteman 1993]. Here an integrated structure and content editor is presented which structures the presentation in terms of a high level abstraction called a *media channel*. These may be specified declaratively in terms of spatial and temporal relationships, and other properties such as text style and graphics presentation. One aim here has been to provide a declarative style of control, where the author declares the properties of and relations between objects and the presentation system sorts out the details of presentation.

An ideal would be to develop a structured system which had the ease of use of a flowchart based system and the clarity of temporal presentation of a timeline based system, while avoiding the multiple views characteristic of many of the systems described. Two of the systems mentioned above merit particular mention here, in that the approach is similar to the one proposed in this work.

MET++ [Ackermann, 1994] is a structured application framework which structures a presentation as a hierarchy of serial and parallel compositions of media items. The presentation is built from time layout objects and media objects, each with a starting point, a duration and an associated virtual timeline. These are composed together in a tree structure

with media objects as leaf nodes and layout objects as intermediate nodes. When the start time or duration of an object is changed the timing of the complete presentation is recalculated. There are several views of the presentation structure, including a timeline representation that charts the x and y position, or other parameters of objects relative to each other.

Eventor [Eun, et al, 1994] presents three different views of the presentation, a temporal synchroniser, a spatial synchroniser and a user interaction builder. Eventor aims to incorporate the characteristics of both time based systems and event based systems (timeline and flowchart based systems) in one authoring system. The system, like that put forward by the author in this work, is based around the Calculus of Communicating Systems [Milner, 1989] as a specification of the behaviour of the system. Given the close relationship between this and the author's work, they will be compared more closely later.

3.3 Content Presentation

Issues of content presentation cover a number of concerns. The first is simply the quality of images and sound presented to the user. This is largely a matter of hardware and systems and coding design, so that little work is separately concerned with the issues specific to hypermedia design. Some years ago, when hypermedia was in its infancy, there was a wealth of reports charting the future hardware, software and coding developments that enabled the technology. The author prepared one in the context of an E.U. funded research project [Newman, 1990], and others

appeared in the literature [Fox, 1991]. Nowadays the technical feasibility of hypermedia is taken for granted and fewer such reports appear. One continuing concern relates to the issues of finding means of presentation that are consistent across a number of hardware and systems platforms. Addressing this issue entails abstracting the presentation away from the structure in some way. This is the aim of the Amsterdam Hypermedia Model [Hardman, Bulteman 1995] and other work [van Rossum et al, 1993].

Improved content presentation has been seen as one means of tackling the navigation problems inherent in unstructured hypermedia. Such research concentrates on finding novel ways of visualising structural or temporal location [Burrill, Kirste, Weiss, 1994] [Newman, 1993][Cypher, Stelzner, 1991], or of presenting visual, or sometimes, audio [Arons, 1991] cues or 3-D binocular presentation to aid location.

An entirely different view of presentation has been taken by Fischer [Fischer, 1997]. He defines presentation as:

> the activity of users who present, to themselves and to others, their understanding of a particular problem through the use of various resources such as technical manuals, diagrams, or conversations.

From this definition it is argued that design of hypermedia systems needs to consider the complete process of information flow from information provider to user, rather than concentration on abstract structural issues. This work is interesting and relevant because it comes from an information design perspective, rather than from the information systems experts who dominate the field. The work included the authoring of an

industrial strength hypermedia technical documentation system using "traditional" authoring tools [Fischer, Richards, 1995].

3.4 Content Structure

The seminal work in defining the structure of hypermedia systems is the Dexter Hypertext Reference Model [Halasz, Schwartz 1990][Halasz, Schwartz 1994]. The Dexter model defines a layered model, following the paradigm of computer system organisation description established by the Open Systems Interconnect (OSI) layered model of communications protocol structure. In Dexter three layers are defined, with interfaces between them as illustrated in figure 3.1.

Layer	Concern
Run Time	Presentation, User Interface, dynamics
Presentation Specifications	
Storage	Database, nodes and links
Anchoring	
Within Component	Structure within nodes

Figure 3.1: The Dexter model

The Dexter model has provided a common basis for the understanding of the structure of hypermedia systems and has formed the basis for classification of work in the field. It is supported by a formal model in Z [Spivey 1989], which provides an unambiguous definition of the model.

One weakness of layer based models is that they can compartmentalise work into a particular layer when the concerns of that work may be better addressed using a more holistic view of the subject, which considers the influence of individual components on the whole. This has indeed occurred with the Dexter model, particularly with the identification of the dynamics of the system with the run-time layer. Many hyperbase systems include dynamic components, such as video clips or scripted segments. According to the Dexter model such data resides within the "Within Component" layer, although its behaviour affects the dynamics of the system, which is a concern of the "Run Time" layer. This issue crucially affects the present work and will be addressed in more detail later.

The strength of a layered model, and in particular the separation of concerns of storage with those of presentation and node content is that it makes the integration of diverse pieces of information together into something that has the appearance of a single system very much easier. The layered structure abstracts away to different layers those aspects of content and structure that do not concern the purpose of a particular layer. At the level of any particular layer the problem of integration is simplified because the limited domain of that layer restricts the range of objects and behaviours that must be integrated. It is possible to integrate within a particular layer without consideration of the content of other layers. In the case of the Dexter model, integration of heterogeneous content to a common presentation can occur at the storage or the run-time layers.

The first approach is typical of the many browser systems for the World Wide Web [Berners-Lee et. al. 1992] that are available today. The Web was developed as a means of distributing hypertext documents to the research community at CERN and was developed on top of the existing Internet services. As such it has had to cater for the multiplicity of services and protocols that already exist on the Internet. Web browsers typically have to deal with FTP [Bhushan, 1972], Gopher [Albertini et. al.], POP [Myers, 1994] and other existing protocols as well as that more usually associated with the Web, HTTP. Web browsers must contain integral support for the component encodings included within HTML documents [Berners-Lee, Conolly, 1995], including several types of image and animation files. As new document types are defined the browsers can be extended using *plug-ins*, helper software packages which extend the capability of the browser. The cost of this approach is that integration of these additional services is dependent on the hypermedia functionality of the plug-ins. If they do allow hyperlink navigation then the integration of the new media reaches a full stop unless the plug in itself includes the functionality to handle any media type encountered at the end of the link. Such contingencies can be handled by adopting a component based architecture which allows the plug-in to use the resources of the browser for navigation.

The second approach is to include the integration in the storage layer. This has been the approach taken by many "open" hypermedia systems. Such systems are open in the sense that information may be included from

a number of different sources and stored using several encodings. They may depend on the use of compatible browser/viewers and are therefore not necessarily open at the run-time level. Some of these systems conform to a link based network model of hypermedia structure while others have imposed a more structured database modelled structure. The latter are dealt with in the next section.

Link service and management systems include Intermedia (Xerox PARC) [Haan et. al. 1992], Sun Link Service (Sun Microsystems) [Pearl, 1991], Multicard (INRIA) [Rizk, Sauter, 1992], PROXHY [Kacmar, Leggett, 1991], Chimera (University of Irvine) [Anderson et. al. 1994], SP3 (Texas A & M University) [Leggett, Schnase, 1991] and Microcosm (Southampton University) [Fountain et. al. 1990].

These systems were generally developed before, or in parallel with, the World Wide Web. Although several of them have distinct advantages over the structure and organisation of the Web, none have managed to maintain their position in the face of the overwhelming uptake of Web technology and its spread from the Internet to intranets. These systems are generally "end-to-end", that is they require specific software to handle both server and viewer, and sometimes specialised authoring tools as well. Thus, although they proclaim openness, and indeed many are open in terms of data formats handled, they appear closed compared with the plurality of browser, server and authoring software associated with the World Wide Web. As a result of this, and withdrawal of vital software

components, several of these systems, including Intermedia and Sun Link Service, are no longer being developed.

Link service systems are generically separated from the World Wide Web, which in this context means an HTML based system, in that they separate content and structure, as opposed to HTML systems, which have links embedded in the data. The consequence of embedding structure information in the documents is that documents become specific to one application, which limits the reuse and multiple use of data sources. Moreover, the database for the system is constrained to exist in the format that defines the embedded links, namely HTML. To overcome this there is a growing tendency on web sites for HTML to form only the framework within which other data is held, allowing helper applications to view the embedded non-HTML data. This data is then not integrated with the hypertext structure and navigation directly from views of this data ceases to be possible. Lately some common document formats, such as Microsoft Word and other proprietary formats have included hypertext links to overcome this problem, but at best this must end up as an untidy and inelegant solution. Use of a link service would have avoided this necessity by abstracting the links away from the content. The goal of an open link server is to interface one or more viewing tools with a heterogeneous collection of content objects. The way this function relates to the Dexter model is shown in figure 3.2. The openness of the system is achieved by designing the link service to handle components with a number of different structures and formats.

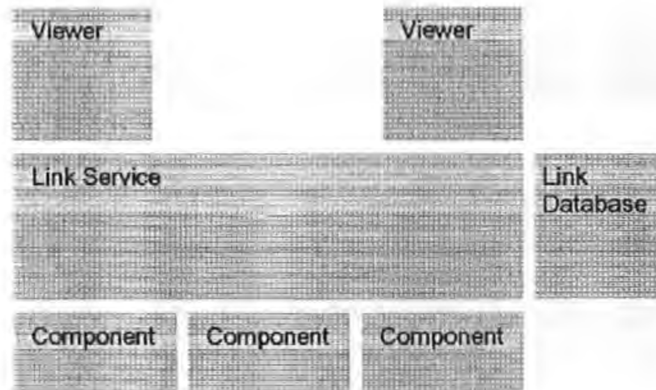


Figure 3.2: Link service related to the Dexter model

The key factor in the production of a successful link service is the maintenance of coherence between the links database, in whatever form it takes, with the component data. This is rendered more difficult if a high degree of openness and extensibility is built into the system. The design of the system becomes a formidable data modelling problem. This is reflected in the large amount of research into data modelling for hypermedia systems that has been carried out both within and separated from the projects discussed above.

3.5 Database Organisation

Link servers conform to the classic network structured link based model of hypermedia. One problem with such a structure is that it is very easy for the user to get "lost" within this unstructured system [Conklin, 1987][Edwards, Hardman, 1989][de Young, 1990]. An obvious solution to

this is to provide a greater level of structure to the system. The data modelling work described in the previous section leads quite naturally to the use of database models to structure the system, as opposed to a network of links. Such systems are similar to the link servers described above, but in place of a links database use structure information based on a database model, with relational or object oriented models being the favourites. The relationship with the Dexter model is illustrated in figure 3.3. Such systems, often called "hyperbase" systems, include Hyper-G (Technical University of Graz) [Knappe et. al. 1993], DeVise Hypermedia or DHM (Aarhus University) [Gronbaek, Trigg, 1992], HB3 (Texas A & M University) [Leggett, Schnase 1994, and HyperDisco] (Aalborg University/Texas A & M University) [Wijl, Leggett, 1996].

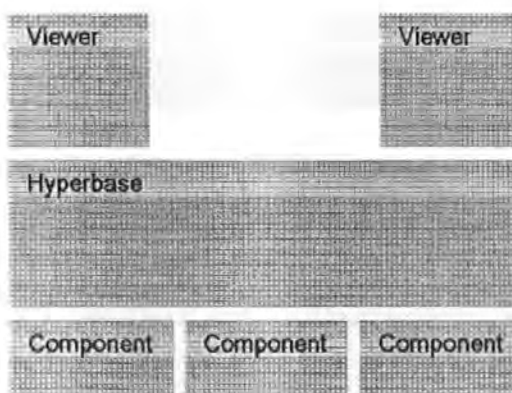


Figure 3.3: Hyperbase system referred to the Dexter model

These systems have tended to suffer from the hegemony of the Web and HTML in the same way that the link servers have. The ability to impose higher levels of structure on the information base does, however, give an

important competitive advantage when compared to unstructured systems. At least one of them, Hyper-G, has been adapted to take advantage of this. The original specialised data formats interfacing from the data manager to the data storage and the viewer have been replaced by HTML, so that the documents are stored in HTML format and the server generates HTML on the fly to feed the viewers. The result is a system that can be integrated with the Web in cases where additional robustness is required.

The understanding and imposition of structure in hypermedia systems has been and will continue to be a fertile area of research. This work has been driven by several concerns. First, the need to understand the properties of the data objects that make up hypermedia systems. This requirement was to the fore in the production of the Dexter Reference Model. The Hypertext Abstract Machine, HAM [Cambell, Goodman 1988], in which an abstract machine plays the part of the conceptual schema, providing an operational semantics for hypertext, was one of the earliest attempts to produce such a model. Its age is reflected in a lack of flexibility, openness and expandability. HB1 [Schnase et. al. 1993], the predecessor to HB3 described above, was based on a sophisticated data model describing the relations between media objects, anchors and links. The MD data model [Gu,Heuhold, 1993] provides data schema describing the conceptual, logical and layout structure of a multimedia document. It describes the structure of the content, the layout structure and the extent of their occurrence in space and time. The VIMSYS data model[Gupta 1991] deals with the storage of images, providing support for information derived

directly from the image. Hypermedia DBMS [Pruckler, Schreff, 1996] uses a quadruple schema system, separating presentation, content, media and storage schema in order to separate concerns of content with those of physical data content. SNITCH [Mayfield, Nicholas, 1993] adopts a similar, though simpler data model.

Data modelling is also central to another strand of open hypermedia work. Here the objective is to provide filters which translate between the data formats used in objects derived from different sources and a database system which stores the data in a common format. This approach is similar to the STEP data interchange formats used for the shared use of data between CAD systems, a very similar application domain. Step is based around a data modelling meta-language (Express) which is used to define the interface to the file content for the different application functionalities which are to be shared. These defined application interfaces, termed "application protocols", or APs, are also included in the standard, although many have still to be defined*. This conversion directed data modelling is central to the Hypermedata project[Cook et. al. 1996]. The AMOS system [Boll,Lohr, 1996] provides a Hyperbase server which presents a standard data interface called VML to its clients but can access and convert data stored in SGML, MHEG, OCPN and HyTime formats.

---

* Interestingly, there is a direct link between the two fields. Several CAD vendors, unwilling to wait for the definition of the APs, are integrating their products using Web browsers such as Netscape, with appropriate plug-ins to allow viewing of the different data formats in use.

3.6 Navigation

As noted above, navigation of hypermedia systems can pose problems, especially when the system is large and loosely structured. One approach to this problem, considered above, is to attack the structure issue by imposition of defined, database based or other structures. Another line of approach has been from user-interface considerations, enhancing user interfaces to alleviate the navigation problems. Such navigation tools generally depend on the presentation of structure information in the form of maps, history lists, graphs or guided tours to the user as an aide-memoir to the location within the hyperbase. These methods have been documented by Nielson [Nielson,1990] and many, such as history trails, are now *de rigeur* in any browsing system. These methods require some form of visualisation of the underlying structure of the hyperbase and therefore almost inevitably become entangled in issues of data modelling. Specialised versions of these enhanced interfaces are described related to navigation through movies [Geissler,1996] [Anderson, 1988], speech [Arons, 1991] and video editing [Ueda, 1994].

One direction of approach is *content based retrieval.* The aim here is to provide a database like query interface to unstructured data by associative access to the content. This approach is particularly attractive for those dealing with large amounts of video data and various techniques and data models have been described by several authors [Dimitrova,Golshani, 1994] [Petrakis, Orphanoudakis, 1993] [Wu et. al. 1995].

3.7 Program based organisation

Most of the literature views the organisation of hypermedia structure as a data structuring issue, with declarative structuring techniques applied to temporal as well as spatial and logical structure. An alternative, imperative based view that sees the structure in program type terms is less evident, at least as concerns deep and large-scale structure. Scripting languages have been part of multimedia from the earliest days, but are generally confined to the behaviour of a single object, rather than the hyperbase as a whole. As a result scripting languages are generally small program languages which cannot deal with large scale structure, illustrated by a comparison between Java (a programming language designed for "programming in the large") [Sun 1996] and JavaScript (a scripting language).

A substantial amount of work investigating the application of program like structuring techniques, namely the use of process algebras, to user interface design, including application to hypermedia systems, has been carried out by Johnson and Johnson [Johnson, Johnson 1991].

3.8 Relationships with this work

The majority of workers in the field have chosen to view hypermedia systems as a heterogeneous collection of objects, each with their own internal spatial and possibly temporal structure. This has led to a view of hyperbase structure that is essentially external to the content of the objects within the database. Thus most work has concentrated on database

derived methods of data and temporal modelling. This attitude is valid for many of the types of hypermedia systems seen in the world today, which are essentially libraries of objects, each with its own internal structure. In this world view there is unlikely to be any overall designer of the system, if it is large. There will be a systems designer or designers responsible for designing and imposing some sort of structure on the system, but the content of individual objects is likely to have been designed by independent authors. In a sense the system is designed and the individual content objects are authored. The field of hypermedia studied in this work is somewhat specialised - high reliability technical documentation systems. In this field of work the overall dynamic behaviour, in the sense of the ordering of events and actions is important. In this context the system designers must be aware, and be able to verify, the dynamic behaviour of individual objects, including any scripts contained in them. This necessitates a top-down, integrated approach to the design of the system, and a design method must cover the structure and temporal behaviour of the whole system, from overall database structure to the behaviour of individual interactive scripts at the leaves of the structure graph.

Such considerations go along with an understanding that the producers of such systems must act as designers of the system as a whole, rather than just collators of information. The work of Fischer [Fischer 1997], an information designer rather than a technologist, is illuminating on this issue. Fischer's thesis argues that even in cases in which the system is "designed", rather than "collected", typically the separation of expertise of

the system designers and the experts who are using the medium to present their information can cause systems to be of less than optimal utility for the users. He presents a theory of presentation that views the design process as the presentation of information from expert to user. Even systems that are designed, rather than collected, are not necessarily suitable for the role of useful technical documentation.

Taken in this light much of the current research work on hypermedia structure does not impact on the current work as strongly as might first have appeared to be the case. By separating the small-scale structure from the large-scale structure of the system the system compiler is separated from the authors of the individual objects, and in fact much of the work on open hypermedia has been focussed on achieving just this separation.

The work on temporal properties of hypermedia has generally been directed towards a declarative approach, stemming from the HyTime specification, which was one of the first attempts to provide a format that could describe the temporal relationships within a hypermedia system. Hardman and Bulterman [Hardman, Bulterman 1995] argue strongly for a declarative approach to temporal structuring.

> Two of these approaches (to authoring of hypermedia systems) are to (a) program the presentation in terms of what happens next on the screen and (b) state the timing and layout relations among items declaratively and leave it to an interpreter to derive the actions required. It is this latter approach we take.... Here, the author is protected from having to produce tedious procedural specifications (for example, place this picture on the screen in area A, then play this subtitle in area B), and can concentrate on creating relations among the objects (such as this subtitle goes underneath this picture). This allows greater flexibility in changing both small and large parts of the presentation.

Against this argument one can place as a counter the difficulties that many users of declarative programming languages have had, particularly in applying temporal structure. This opinion is based on the author's experience in teaching of programming and has been discussed in [Newman et. al. 1994] and [Newman et. al 1995]. In approaching this issue the author has chosen to apply his background in software production for real-time systems, a field which is dominated by an imperative, program structured approach. This does not signify any statement about the relative theoretical merits of the two approaches.

In taking the imperative approach, the work of Johnson and Johnson [Johnson, Johnson, 1991] in pursuing the temporal issues in a related field, user interface design, has been directly relevant. So too has the work of Milner [Milner 1989], Hoare [Hoare 1978] and others in the development of process algebras. This work, not being directly related to hypermedia, has not been discussed in this chapter, but is, in some detail, in Chapters Four and Seven.

Chapter 4: A methodology for design of large hypermedia systems

4.1 Introduction

Hypermedia is a young medium and design methods for hypermedia systems are in a relatively early stage of development. Current design methods have followed one of three different paths. They are *Scripting*, based on simplified programming languages that allow definition of the content and sequence of the system; *interactive tools* which allow the system to be constructed by form filling and "programming by example" and *database* methods which treats the system as a database of visual information. This paper proposes a new method for the design of very large, highly interactive, hypermedia databases for which correctness and reliability is a major requirement. Examples of such databases are the technical documentation systems for safety critical systems. When applied to systems of this type existing methods show severe shortcomings. The interactive and scripting methods because they cannot guarantee correctness or reliability and the database methods because they cannot guarantee the required interactive properties. The new method is based on

one of the most successful methodologies for the rigorous design of real time systems software, where the formal description of the system uses process algebras. Real time systems share many basic characteristics with interactive systems and the application of their design methods allows systems to be rigorously designed both with respect to their content and their interactive behaviour. Furthermore, the use of these methods offers the prospect of the formal verification of the operational characteristics of these systems, if not the correctness of their content. It is argued that a design method based on process algebras possesses the necessary properties for large, safety critical documentation systems and also that, if correctly structured, such a method should be accessible to hypermedia designers.

4.2 High reliability large Hypermedia databases

Hypermedia, the combination of hypertext or non-linear text systems and multimedia, is beginning the transition from a hi-tech toy to a more serious medium. As yet the "serious" applications are relatively undemanding (in terms of data complexity, if not in terms of the graphic and media design). Typical applications are marketing, computer aided teaching and catalogue data. These applications remain simple for different reasons.

Marketing or advertising data may have complex interactive content and often quite a rich structure. On the other hand there is no great demand for correctness or ease of access or navigation. The "rich"

structure is in fact unstructured, designed in an ad-hoc manner by the multimedia author without any reference to any particular specification.

Catalogue information is highly structured, but according to well-known and well-understood database structures. Such systems contain little interactivity, the user interaction being limited to the application of a particular search strategy, usually selected from a fairly small list of options.

The design of computer aided teaching material has tended to concentrate on the interactive nature of the medium, and the techniques used are often derived from the film world. Design will usually start with a storyboard, showing a fixed sequence of frames. Selection of alternative sequences is described using a flowchart referring to the different strands of storyboard. These are then realised using an interactive tool such as Director or FrontPage. Again, the level of structure is relatively low, sometimes to the detriment of the more complex material. In general, however, the underlying structure, being based on material with a fairly simple linear sequence of lessons, remains simple and the need for higher levels of structure is not great for many subject areas.

The author has recently participated in a project to define complex multimedia systems suitable for technical documentation [Newman et. al. 1997]. The research undertaken within this project has shown that applications of this type have a much tighter requirement for structure and ordered presentation than those application types given above. Moreover, the access patterns are very much more complex than are those given above. The characteristics of this type of application are discussed below.

There will be many possible navigation routes through a multimedia technical documentation database. As stated in Chapter 2, for maintenance documentation a task-based organisation has been found to be appropriate, in line with earlier research results [Johnson et. al. 1988]. The information is sequenced according to a number of set maintenance tasks. These maintenance procedures are usually set out in the paper-based documentation as flowcharts. The task descriptions in the paper documentation contain references to other relevant material, such as component descriptions or other maintenance procedures. In hypermedia systems these are implemented as links which cut across the task based organisation, since it is possible for the user to follow such a link and then fail to return to the task based sequence.

There are often different ways of using the same database. For instance, as well as supporting maintenance operations, the same information may be used for training, diagnostics and repair. Each of these different activities will require a different navigation path through the documentation. Each such path will have cross links and the overall structure of the database will become unmanageably complex.

A further level of complexity is caused by the need to cater for different variants and maintenance specifications. The required procedures may change depending on the precise build specification or maintenance history of the unit under maintenance. The maintenance procedures can be modified and refined over time as well, so that any given path through the

database may need to be replicated several times to cater for these variations.

The requirements for correctness for such systems are also strict. For safety critical applications it is vital that the maintenance procedures are preserved in the correct sequence and are complete. While this is simple for a straightforward linear sequence, once we take into account the huge number of possible variations and alternative paths through the database great care must be taken in the design and implementation.

4.3 Existing design methods

Existing methods for the design of multimedia systems are based around a number of basic metaphors that dictate the overall approach of the designer. Unsurprisingly, these metaphors reflect the heritage of the user community which developed them, and each can be identified with one of the communities that can claim some kind of ownership of the multimedia field. The use of a particular metaphor gives rise to a method of working related to that metaphor, which in turn influences the priorities and constraints put on the multimedia designer, and ultimately the form and function itself. Mixtures of the metaphors are possible, and nowadays even common, but like most mixed metaphors, they do not always produce a well-structured or elegant product. These metaphors are detailed below.

4.4 Multimedia as document

As word processing and document management systems have become more sophisticated they have begun to grow in the number of different media that can be included within the document. The early word processors had a page layout model based on a typewriter, with fixed pitch fonts and simple page layouts. As printing technology has advanced the documents have begun to include variable width fonts, a multiplicity of different typefaces (commonly all in the same document!), complex page layout including multiple columns, inset text and illustration boxes, tables, charts, line illustrations, photographs, multiple colours and full colour images. Today's word processors have capabilities beginning to approximate more to a typesetting system than a typewriter.

Along with advances in printing technology there have been parallel advances in display technology, allowing the document to be presented on the screen in a fairly close approximation to its printed appearance. Along with the increasing availability of powerful, high quality personal computers, the consequence is that many documents are viewed only on the computer screen, and need never be printed. For such documents there is the opportunity to include non-printable media such as moving illustrations and sound annotation. At this point the document ceases to be simply a document and becomes a multimedia presentation. The document heritage is still very clear, however. The primary component is still the text, and all other structure is dictated by the textual structure of the document. Non-textual media remain very much a subsidiary concern. Illustrations and the

structure remains very much rooted in its paper ancestry, namely a paper based, page orientated structure, which by comparison with other multimedia forms is quite rigid and static.

The tools used to create and manipulate this type of multimedia are typically document creation systems, word processors and desktop publishing systems, which now include the ability to embed non-paper-based media within a document. These tools include word processors such as Microsoft Word, Novell WordPerfect and Lotus Word Pro, document layout systems such as Quark Express, Adobe PageMaker and Xerox Ventura. Tools and standards have also been produced for the storage and distribution of such documents (Adobe Acrobat, etc).

4.5 Multimedia as game

Another multimedia lineage springs from the computer game. Once again, advances in technology have presented games manufacturers with a steady stream of new or improved media to increase the impact of their products. The essential element of a computer game has always been its interactivity. Starting from the first commonly available game, Pong, in the early 1970s, the aim has been to increase the effectiveness and reality of games by using improvements in computer graphics and other media such as sound. Currently games typically make extensive use of 3-D computer graphics, sometimes added to or mixed with video, and sound. Servo-motors are beginning to be included in games playing equipment to provide sensory feedback, acceleration and gravity effects. Binocular 3-D

presentation is being used. Doubtless soon to come, wind, odours and other sensory stimulation. Recently games have been the benchmark by which other computer graphics media have been judged.

Typically games have a very high level of interactivity and a low level of structure. Games players seem willing to tolerate dysfunctional user interfaces, and partially working or failure prone systems in a way that few other computer users would. Traditionally games software has been developed by "hackers", people talented in code production but not necessarily in its construction. Their work has been constrained by the limited nature of the hardware used to run games, and as a result, until recently software engineering methods have rarely been employed to assure the quality of the product. With the increasing use of servo-motors in such equipment it is likely that concerns for the safety of the user in the event of a malfunction will result in the enforcement of a more rigorous development regime.

4.6 Multimedia as movie

Another way of looking at interactive multimedia systems, particularly those in the entertainment industry, is as an extension of the movie, allowing the added feature of audience participation. To an extent this view is an extension of the games view, but the history is different. Influences include the experience of use of computer graphics for special effects in the film industry and the move of film producers and distributors into the video and video games market.

4.7 Multimedia as database

To those trained in information systems it is natural to view a multimedia system as a database. A database is simply an organised collection of records, and if the records happen to include data that could be viewed as a picture or a movie or some other medium then you have a multimedia system. The advantage of the database view of the world is that there is a very well developed theory and methodology for the construction of very large, reliable, well-ordered databases. Much of the modern world's data systems depend on such databases. To the database designer the content of the database is immaterial; the structures will work whatever the content. What is more problematic is how the material is accessed. In database terms, the query mechanism. Database query methods are based on textual records, and so long as the database contains text linked to the non-textual data then such well understood mechanisms can be used. When it doesn't, query methods based on some other record content will have to be used, and there is much active research on this topic.

There are other issues associated with multimedia databases. One is that of providing query tools that allow access to the different media – most database tools have a very basic textual presentation mechanism. There are several different approaches to this issue. One is to produce a query tool which includes presentation mechanisms for the different media (such as, for instance DHM [Gronbaek, Trigg, 1992]), another is to use the database system as a "back-end" which orders the data which is presented by some other systems, such as a web browser. Hyper-G [Knappe et. al.

1993], for example, has evolved into a system of this type in order to integrate with an HTML dominated world. Another is an object oriented approach, where the records themselves contain or imply the presentation mechanism, using some standardised way of distributing presentation programs such as Java.

The properties of database based systems are derived directly from database systems themselves. While the contents of the database will be well ordered, they will be ordered only in the manner intended by the designers. Typically a the design of a database will not include sequence or other temporal properties, and there is unlikely to be any interactive behaviour except that provided by the query mechanism. Thus a database is liable to be a fairly static, non-interactive data repository.

4.8 Multimedia as program

The other branch of the computer science profession is that of the program designers and software engineers. For these people it is natural to see any system that includes interactivity and adaptive response to user input as a program. The activity that other multimedia designers see as 'scripting' will be viewed by software designers as 'programming', and the design of multimedia systems as program design. This is evidenced by a look at the various scripting languages used for the production of multimedia systems. These are clearly derived from and in most cases still are, programming languages. The fact that they are simple languages fits well with the simplicity, when viewed as a program, of most current

multimedia applications. Real computer programs, on the other hand, need real programming languages and real software engineering methods to produce them. However, many computer programs, particularly computer graphics programs such as CAD and visualisation programs, may be viewed as multimedia applications in their own right. With multimedia applications becoming increasingly complex it is perhaps fair to ask why "real" program development methods shouldn't be used for them as well. Currently such methods are accessible only to computer science professionals (who would probably favour such a limitation), and also are unlikely to have the productivity required to produce technical documentation on a serious scale.

4.9 Multimedia as hypermedia

The final view of multimedia systems is derived from the hypertext model. Hypertext is derived from computer-based training or help systems, where words or phrases in a text can be linked to some other piece of text. The connections between the text form a graph that can be navigated by selecting the links in each piece of text. Interest in hypertext as an organisational model for multimedia systems has been high because it seems to overcome the structural limitations of traditional media. In fact, any structure that can be represented as a graph is possible. Hypertext systems have now gained access to other media to become hypermedia systems, still with the same link based structure. The best known and most widely used hypermedia system is the World Wide Web, and many other multimedia systems have adopted the same organisational model. These

systems are so well known that for some hypermedia and multimedia have become synonymous, although as discussed here, other organisations for multimedia systems are possible.

4.10 Design choices for the hypermedia designer

All of the paradigms discussed above have met with success as models for the design of multimedia systems. The aim here is to select a model suitable for "serious" multimedia applications, namely large scale, high reliability technical documentation. Considered in this light, the choice of design paradigm is rather different. That chosen must be capable of supporting rigorous design methodologies that will allow the production of high quality documentation systems with confidence. This constraint rules out the game, movie and hypertext models, simply because such methodologies have not been developed for these models.

The remaining paradigms are those of document, program and database. The document model may be ruled out because although there are established principles and procedures for quality assurance of documents they are not based on any mathematical idea of correctness. They are unlikely therefore to be susceptible to computerisation, which will be necessary if the required productivity is to be achieved. This leaves the two models rooted in computer science, the database and the program model. Both come with well-established bodies of theory and practice aimed at ensuring correctness. Between these the choice will have to be made according to performance issues.

Most current work on large multimedia systems has concentrated on database models. There are two reasons for this. Firstly most of that work has concentrated on the requirements of large data repositories, rather than on very clearly goal directed systems which also happen to be large. It is this factor which makes interactive performance much more important in the case of technical documentation, and leads to the conclusion that the program based structure is superior. The second reason is that database design and planning methods are in many ways more mature than formal methods for the design of programs. Whereas almost all serious database systems are formally designed and validated, the penetration of formal methods into software design is much smaller. Although most large software systems are designed using "semi-formal" methodologies, that is methods which have some sort of formalised method of working but are not based on any mathematical rigour, fully formal methods have a smaller, although growing, level of acceptance. This is partially due to the low level of accessibility of the methods to those who have not been trained in the mathematics that underpins them. In some cases this causes an alienation from the methods which can amount to outright hostility. Another reason is that when it comes to large-scale systems many of these methods have still to prove their capability. They are ultimately based on the concept of mathematical proof; in real-life systems the size and complexity of the theorems that must be proved lie outside the capabilities of the average human mind.

This discussion seems to be leading to the conclusion that the program paradigm for multimedia systems is also unsuitable for large high reliability systems, but the arguments above can be countered by several other arguments. Firstly, ways are being found of making the operation of formal methods easier. Such things as specification editors, proof assistants and editors, and integrated specification tools can make the task tractable and have become accepted, indeed required, in some safety critical industries. Secondly, production of systems of this scale is a multidisciplinary exercise. Undoubtedly there is a requirement for people with the correct type of formal mathematical training to undertake the verification work, but that does not dictate that these people must form the whole team. So long as notations can be devised that allow the transfer of ideas between different parts of the team, that are accessible to the whole team and which can fulfil the requirements both of the creative and analytical side of the work then the scenario remains viable.

With this in mind it is suggested that program based organisation should be considered as a suitable model for the design and construction of large, high-reliability hypermedia systems. To enable this a methodology is required which can encompass the entire development team. A notations is required which will allow individuals working on development to communicate with and also support the formalism required to assure the quality of the product. The rest of this chapter will put forward such a notation and describe the methods it supports.

4.11 Methods and notations

The previous section suggests that a good starting place in the search for a methodology for multimedia system design would be formal notations for programs. In order to begin the development of his methodology two questions need to be answered. The are "what kind of programs are hypermedia systems?" and "what kind of method is best suited to the development of this type of program"?

To answer the first of these we need to consider which kind of program best aligns with multimedia systems. As was argued above interactive multimedia systems are essentially programs, in that the system defines the sequence or control flow of a series of events in the same way that a computer program does. This identification immediately suggests one particular programming paradigm, the *imperative* paradigm as opposed to the various *applicative* paradigms. This is underlined by examination of the various scripting languages used in multimedia authoring systems: all are imperative. One could speculate on the practicality of a functional scripting language – it is certainly theoretically possible, but would be difficult for those not familiar with such languages to grasp or to program. It is difficult to find examples of the successful programming of interactive systems using functional languages. The most commonly cited example of such a case is the use of the language Lisp [McCarthy, 1960] as the base programming language for the Symbolics Lisp Machine or as the macro language for the AutoCAD CAD program. However, Lisp is not a pure functional language since it includes variables and assignment, introduced

into the language precisely to simplify the data handling surrounding sequential events.

Systems modelled on the database paradigm will tend to be designed using data modelling languages and accessed using query languages. Both are applicative and as a consequence (or maybe *vice-versa*) the systems are not substantially interactive.

The imperative paradigm is also a good match for the subject area. Many of the procedures involved in maintenance, repair and diagnostics are defined as step by step lists of instructions. It is not unusual to see them expressed in manuals as flowcharts – a notation which originated in programming but is now, ironically, rare in that field. Thus we might expect that an imperative model would be accessible to the various people involved in authoring technical documentation.

Advanced multimedia systems go beyond the behaviour describable using a simple imperative programming language. In particular it is common to have different things happening on different parts of the screen. A look at many pages on the World Wide Web will show them to be full of animated images, to contain "buttons" which are not simply links but spawn new browsers which can view other pages while the original remains displayed. In short, these system are concurrent systems. A look at the languages used to implement them reinforces this view. Java, the language most often put forward as the language of advanced distributed multimedia systems, is a concurrent programming language.

Thus it is likely that we will find the inspiration for multimedia design methods in the methods used for concurrent software systems. Reassuringly, this is also the body of knowledge that covers the design of real time and embedded systems. Such software systems are used in computer controlled interactive products and these are often safety critical applications. Due to the need for assured software quality in these applications, the theory of concurrent programming and the design methods that go with it is well advanced.

We will now proceed to consider the next question: what kind of method would be suitable? One theory of program correctness for imperative programs is derived from the *predicate calculus*. Predicates are associated with the state of the system before and after execution of each part of the program and the job of the program proof is to show that the execution of the program transforms the first predicate, the *precondition*, into the second, the *postcondition*. In this view of the world the role of the specification is to define the required precondition and postcondition, to say *what* he program must do, rather than how it must do it. The drafting of such specifications requires a notation that contains the necessary symbols and operators for the operation of formal logic on the specifications, since formal logic is the mechanism by which the predicate transformations are demonstrated. Such notations are called specification languages, typified by Z [Spivey, 1989] and VDM [Jones, 1990]. These are possible candidates as the starting point for the development of a notation for multimedia design.

This view of a program is essentially a "batch process" view of the program. All that is important about it is how it starts off and what it produces, what it does in between the specified start and end points is unimportant. In interactive systems and other real time systems what happens in the middle is vitally important, because it is these events that define the interaction with the system. As far as the precondition and postcondition of a program is concerned the ordering of two events is immaterial, on the other hand, as far as the user is concerned, if the two events occur in the incorrect order the system may be unusable. One way to produce specifications which cater for this is to introduce predicates, which describe the intermediate states before and after each necessary event, and to prove that the program transforms one to the next in the correct sequence. The specification is now a set of predicates that define these states and the order in which they must occur. Such methods have been developed and applied with some success to the formal design and specification of concurrent programs [Andrews, 1991]. This approach is called a *state based* specification method, since it seeks to describe the ordering of the system by defining the sequence of states that it must go through.

An alternative approach is an *event based* one. Here it is acknowledged that the events themselves are of primary importance and as a consequence how the job is done is as important as the end result. In an event based model the specification specifies which events the system will enter into and in which order. As a result of this event based

specification languages look very like programming languages – in both cases they seek to describe a (possibly flexible and adaptive) ordering of events – and many of the concepts, constructs and operators are very similar. This has advantages and disadvantages. On the one hand it has already been noted that imperative programming languages seem to be easily adopted by many people, and at least superficially, it is the same with event based specification methods. Most people can write down the order in which they think things should happen. On the other hand it can be very difficult to maintain clarity about what level of abstraction is in use at any time. When the specification language looks very like a program it can sometimes seamlessly evolve into that program, and one is left wondering whether a formal specification ever really existed.

Of course, the essence of a formal method is that it is amenable to formal analysis, and this is where the event based specification differs from a program. As well as a specification language it is an algebra, more specifically a *process algebra*, the symbols and formulae of which describe processes of events and the rules of which allow the formulae to be manipulated to analyse those processes. Using this algebra a calculus can be constructed allowing the analysis and verification of systems of processes. A *semantic model* that describes the meaning of the symbols and the way the rules are applied underpins the algebra. As long as one believes that the semantic model aligns with reality then the process algebra forms a sound basis for analysis of the properties of the system being designed. The suitability of process algebras for design of structured

dialogues has been investigated previously, notably by Alexander [Alexander, 1990] and Johnson [Johnson, 1991]. Application to hypermedia is a natural extension.

Compared with the state based specification methods the process algebra approach has advantages and disadvantages. As noted above it is more readily accepted by a larger group of people, at least superficially, than the more esoteric predicate systems. As a system based on events it is a much more natural fit to event based products such as multimedia systems. Intellectually it is certainly much easier to specify the sequence of events that needs to occur than it is to convince oneself that everything that needs to be said about a particular state has been expressed in the predicate which is supposed to specify it. However, the ease of specification is not matched by a corresponding ease of analysis. Process algebras have a lot of rules. Far more than, for instance, Boolean algebra with which many people are familiar. Constructing proofs in Boolean algebra can be hard enough as it is, and those in process algebras are more difficult still. It is unlikely that it will be possible to prove completely any complex system. Instead the aim is to define *safety* properties (things that must not happen) and *liveness* properties (that the system will operate). It seems to be much easier to frame these properties as predicates that as sequences of events.

For the domain of hypermedia technical documentation the positive features of process algebras seem quite compelling. Not only do they match the requirements of the area well, but they have at least a chance of

being adopted by a community which has taken to scripting languages readily enough. A further refinement in their use would be to adopt a semi-graphical notation, still consistent with the underlying structure of a process algebra, which also has some commonality with the methods used by the "creative" people in the authoring process, namely the graphic artists and information designers.

4.12 Selecting a process algebra

There are a number of different process algebras that have been developed each with its own proponents. Each is based around broadly similar concepts, although the notation for each and therefore the "look and feel" is very different.

The earliest process algebra was Hoare's Communicating Sequential Processes (CSP). CSP [Hoare, 1978] introduced the notion of a process as a sequence of events, with a rather elegant recursive definition of a process as an event leading to a process. The other notion introduced by CSP was a model of processes transferring information between themselves in a simple, ordered, synchronised way. This simplification of the inter process communication model allowed the issues of transfer of information between processes and synchronisation between them to be handled without recourse to shared variables, allowing a huge simplification of the analytical apparatus needed. CSP has a very terse, mathematical syntax with a wealth of unusual symbols denoting operators and standard events and processes, which can be daunting to those not comfortable with

mathematical notations. CSP has been developed into a range of different variants, including Timed CSP, Receptive Process theory (RPT), and Theory of Asynchronous Processes. For our purposes CSP is adequate. There are also a number of semantic models that have been developed to underpin CSP. Again, for our purposes the simplest, the traces model, will be sufficient.

LOTOS [van Eijk et. al., 1989], or Language of Temporal Ordering Specification, is a process algebra proposed by the International Standards Organisation as an international standard for the specification of concurrent and real time systems. LOTOS has a Pascal like syntax which makes it appear very much like a programming language, improving its user-friendliness to the programming community, but making it very much more complex to manipulate.

CCS, Calculus of Communicating Systems, was developed in 1989 by Milner [Milner, 1989]. This language shares many common features with CSP, but is notationally quite different, although still highly "mathematical" in appearance. CCS is a simpler algebra than CSP, lacking any concept of data, among other things. CCS specifications concentrate on events and their ordering, whereas CSP can say something about data values as well. As a result of this simplification the rules and verification of CCS specifications is simpler. There is an associated logic, Hennesey-Milner Logic [Stirling, 1991], or HML, which can be used for reasoning about specifications. The semantics of CCS is based on an operational semantic model. CCS has also been developed into a family of languages, but there

is much more semantic commonality between them than is the case with the variants of CSP

For the purposes of specifying the sequence of events in interactive multimedia systems any of the above would be suitable. CSP suffers from having been developed before the semantic models were fully developed. Different workers have developed different semantic models, and to accommodate them the language has grown, with many operators, the subtle differences of which are apparent only with reference to a particular semantic model. By contrast CCS has had a well-developed semantic model from the start, and this concentration on one or, more accurately, two equivalent semantic models has allowed the language to remain tiny in comparison with CSP. CCS has the advantage that it concentrates on the nub of the problem which concerns us, the ordering of events, and therefore is more closely optimised to this particular application. An earlier version of the notation was based on CSP. The notation was very much more complex than the one presented here, and it was not at all clear that it allowed any greater expressivity. The author was required to distinguish between the subtly different nuances of CSP semantics, for instance between "demonic" and "non-demonic" choice. The adoption of CCS brought a great simplification, to the point where the apparent simplicity of the notation belies its power.

CCS also has the advantage of a good range of available support tools and quite an amount of active research associated with it, reviewed,

for instance, in [Fencott, 1996], so that the body of knowledge concerned with CCS continues to grow.

4.13 Dealing with size and complexity

Formal specification systems such as a process algebra tend to be best at handling small systems. As soon as the system reaches any size then the increase in size of the state space of the system makes any sort of reasoning about it very complex, even with the help of computerised reasoning assistance. This is precisely the same problem as that of keeping track of the design of large programs. Even though the design notation works at a level of abstraction that strips away the irrelevant detail, the amount of relevant detail can still be too much to be handled easily.

The solution to this problem is to allow a way of raising the level of abstraction still further, that is to separate the system into component parts and at the top level of design to concentrate on the component parts and their interface. What occurs internally to those component parts is irrelevant so long as the part operates correctly and can be viewed as a "black box", so that the internal workings are need not be visible.

To design a system in this way requires two conditions to be fulfilled. One is that the design notation is modular or hierarchical in style, and can support this abstraction process. Process algebras partially meet this condition, since processes can be composed to form other processes, and a process can be used to form the basic building block of the system. Where they fail fully to meet the requirement is with respect to the internal

invisibility requirement. In most process algebras all names are visible throughout the system, which in a large system produces the likelihood of the phenomenon known as "unintentional capture", or more simply dual use of the same name for different purposes. To overcome this they allow an operation known as "hiding" where a name is hidden from the outside world to allow it to be used again without fear of capture. Modular programming languages, by contrast, "hide" names by default. To make them visible to the outside world requires explicit use of an "export" operation. In devising a methodology for multimedia systems design using process algebras one could either accept the additional need for discipline on the part of the designer that this condition imposes, maybe with some help from the design tools, or to adapt process algebras to follow the "programming" model. If existing process algebra tools are to be used in the methodology then the former is probably preferable.

The second condition is that the designers conceive and structure the system in such a hierarchical, top down fashion. This tends not to be the case at the moment, but this is partially due to the use of unstructured design methods. Experience of software engineering methodologies suggests that as structured methods are introduced software designers have become used to designing their software in a structured way, even if it is less "intuitive". It is to be hoped that the same will be true in the field of multimedia systems design.

4.14 Framing safety conditions

The other important part of the specification is an account of the features critical for the success of the system. In concurrent system design parlance these are the *safety conditions*. The safety conditions define the conditions that must be present for the system to operate safely. Often a safety condition is defined as a negation of some condition that must not be present if safety is to be maintained. In the context of multimedia documentation they are likely to be concerned with sequence or completeness of presentation. A critical sequence must be presented in its entirety and a sequence of critical assembly or disassembly operations cannot be presented in the wrong order.

There are two crucial issues in identifying the safety conditions. The first is identifying the conditions themselves. In a large system correctly identifying the critical areas of the domain will be no trivial task. It will always remain subject to domain expertise, and is a part of the job that no formal method can help with. The fact that a method demands that these critical areas be identified might, however, help focus the minds of the domain experts on analysis of the usage of their documentation.

The second issue is the translation of these conditions into a formalism suitable for use in analysis. The formalism associated with CCS is Hennesey-Milner Logic (HML) and its extensions. This process has two parts. The first is expressing the activity which comprises the condition as an agent in CCS or its semi-graphical equivalent. The second is composing a predicate in HML which expresses the condition whether or not this agent

exists within the correct set of temporal and state conditions. Although HML is not as complex some formal logics, use of such logics is likely to remain the domain of trained logicians. While there is much mileage in developing a designer friendly, graphical notation for the process algebra itself, this is probably not the case for the associated logic. The values of the logic are the agents and events of the process algebra, and these may be specified using a graphical notation. Manipulation of these values by mathematicians will be achieved more easily in a classical mathematical notation.

4.15 Applying process algebras to hypermedia systems

The value of a design method based on a formalism is not necessarily that everything will be proved to be correct. It is more that modes of design are encouraged which lend themselves to be provably correct, and are therefore more likely to be correct. In particular, the freedom of the designer is constrained in such a way that structures which are likely to produce unanaiysable (and therefore unprovable) results, are not permitted. The task for hypermedia system design is to define a set of design primitives that satisfy this requirement, but still allow sufficient expressivity to allow definition of systems with the required characteristics. Such a set of primitives exists within the realm of process algebras. Process algebras, such as CCS, model the world as systems of co-operating sequential processes This forms a suitable starting point for the hypermedia design method.

The basic entities in process algebras are *events* and *processes*. In CCS a process, more usually called an *agent* is simply a sequence of events.

$$P \stackrel{def}{=} e_1.e_2.e_3\ldots$$

where e_1, e_2, e_3 are events. The '.', or prefix, operator indicates the sequence of events, that the event on the left hand side precedes the event or agent on the right hand side. The '=' with the word 'def' above indicates a definition. The mapping of these concepts to hypermedia design is simple. The event corresponds to a user interaction (clicking on a button or link) or display (or sound) output, the process corresponds to the sequence of inputs and presentation that results. Since the definition of a process can include any number of events it can include any further events, leading to other processes, corresponding to the interaction points included within those pages. A short example, of an agent to provide interactive help, is given below.

$$Help \stackrel{def}{=} helpscreen\ .helpbutton\ .HelpTopic$$

This definition indicates that the process or agent $Help$ consists of the event $helpscreen$, which we might interpret as the display of a help screen containing a button marked "help" which the user presses to get help. This is followed by an event $helpbutton$, corresponding to the pressing of the button. After this comes the invocation of an agent called $HelpTopic$, which

will consist of a further sequence of events and agents. It should be noted that in CCS there is no distinction between "input" and "output" events.

It is necessary for a page to include a number of different action points. If we allow the user to select only one at a time then we can use the CCS choice operator to signify the available choices. The form of the choice operator is as follows.

$$P_1 + P_2$$

It generates a process which behaves as P_1 if *its* first event (button press) occurs and P_2 if that process' first event occurs. An example of the use of the '+' operator is as follows.

$$HelpMenu \stackrel{def}{=} helpscreen\ .(button\ 1.Topic\ 1 + button\ 2.Topic\ 2)$$

This indicates that the *HelpMenu* is defined as the display of the help screen, as before, but now there follows a choice, dependent on whether *button1* or *button2* is pressed first. *Button1* invokes the *Topic1* agent, *button2* invokes the *Topic2* agent. This definition of *HelpMenu* allows a single choice to be made, and then terminates. We can use a recursive definition to allow the menu to run continuously.

$$HelpMenu\ 2 \stackrel{def}{=} helpscreen\ .(button\ 1.Topic\ 1 + button\ 2.Topic\ 2).HelpMenu\ 2$$

There are two possible ways of sequencing the pages. Pages can either follow on from each other sequentially or else can be opened in a new window, and can continue concurrently with the original page. These situations are handled by the CCS sequential and parallel composition operators. The '.' or *prefix* operator, is used as follows.

$$P_1 . P_2$$

This indicates two pages following on sequentially. The operator is the same as the prefix operator seen previously. Since an agent is simply a set of events, so the prefix operator may separate two agents as well as an event and an agent. The parallel composition operator is used as follows.

$$P_1 | P_2$$

This indicates two pages displayed simultaneously. This state of affairs occurs when a button invokes its agent in a new pop-up window, rather than replacing the screen contents. The process *HelpMenu3* below operates in this fashion.

$$HelpMenu\ 3 \stackrel{def}{=}$$
$$helpscreen\ .(button\ 1.(PTopic\ 1 | HelpMenu\ 3) + button\ 2.(PTopic\ 2 | HelpMenu\ 3))$$

Sometimes it will be necessary for one process to cause some effect on another. In the following example the buttons control a video player window, which has two simple controls, start and stop.

$$VideoPlayer \stackrel{def}{=}$$
$$videoscreen.(startbutton.(Play|VideoPlayer) + stopbutton.\overline{stop}.VideoPlayer)$$
$$Play \stackrel{def}{=} nextframe.Play + stop.\mathbf{0}$$

In this example *VideoPlayer* displays the control screen and then either invokes the process *Play* in parallel with a recursive invocation of *VideoPlayer* if *startbutton* occurs or, if *stopbutton* occurs, it "outputs" the communication event *stop* with an overbar. This event has a complementary event *stop* without the overbar, the two of them together represent synchronisation using the named channel *stop*. The video player is defined by the agent *Play*, which either displays the next frame in the sequence or, if the event *stop* occurs, does nothing and terminates. "Doing nothing" is symbolised by the primitive process *0*.

In the context of the design of large systems, and particularly to allow reasoning about equivalence between agents, we will need to use the CCS *hiding* operator, which hides a name from external view. Use of this operator allows reuse of components that make common use of names, by hiding those names form each other. In the textual syntax of CCS hiding is indicated by a `\`, so

$$Q \stackrel{def}{=} P \backslash name$$

indicates that Q is defined to be the same as the agent P with the name *name* hidden from external view.

This set of entities and operators is all that is needed for the majority of hypermedia systems. It should be noted that only the event sequence, not the content of the pages is being described. The content could be text, images, diagrams or continuous sound or animation without affecting the basic structure. Means of defining the content will be suggested later in this paper.

4.16 A graphical notation

In the raw mathematical form put forward above process algebras are unlikely to be acceptable to practising multimedia designers. What is required is a more accessible graphical design notation that can be simply translated to algebraic form if required for analysis and proof.

The notation described below has been designed so as to maintain a form familiar to authors while at the same time maintaining a strict one to one mapping with the CCS notation above. The construction rules also obey the construction rules of CCS. The graphical notation is designed to be drawn on a grid of frames, where each frame may contain a story board sketch.

Each frame may represent an event, or a process invocation. The bulk of the frame is given over to the storyboard sketch, the frames are labelled at the bottom to identify them. The names follow normal CCS conventions with agent (process) names starting with an upper case letter and event names starting with a lower case letter. A definition is indicated by a name contained in box over, rather than under, the frames.

Sequential composition (the '.' operator) is denoted by juxtaposition from left to right and parallel composition (the '|' operator) or choice (the '+' operator) by juxtaposition from top to bottom. Parallel composition is denoted by a vertical bar running down the left side of the frame, choice by a '+' symbol in the centre of the left border of the frame.

Names are hidden by shading in the name box. Where a name appears more than once in a definition a single shading hides every occurrence.

A process definition is shown in Figure 4.1.

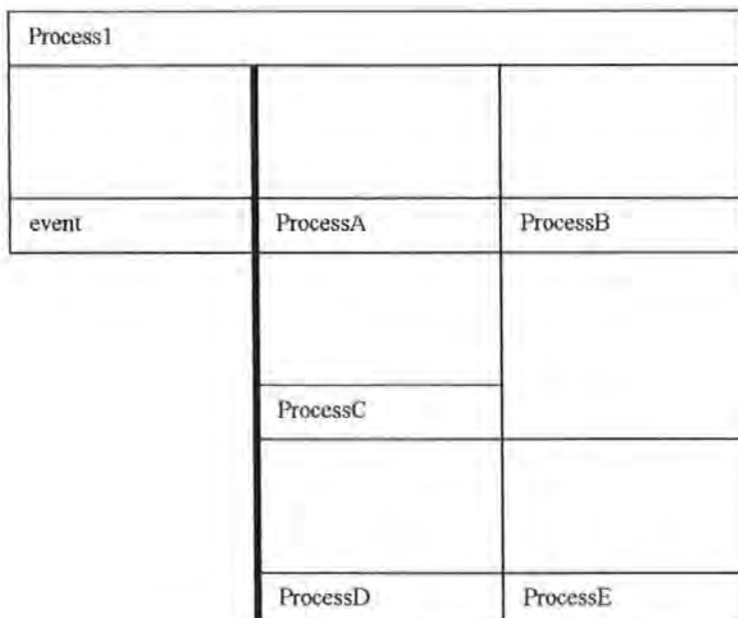


Figure 4.1: A graphical notation

This defines Process1 to be event followed by parallel invocations of *ProcessA* followed by *ProcessB*, *ProcessC* and *ProcessD* followed by *ProcessE*. In CCS this definition would be equivalent to

$$Process \stackrel{def}{=} event.(ProcessA.ProcessB)|ProcessC|(ProcessD.ProcessE)$$

In the case where we are describing only the composition of processes we may decide to do away with the storyboard boxes to make the layout more compact. The second example, shown in Figure 4.2 is the *VideoPlayer* process translated into this semi-graphical notation.

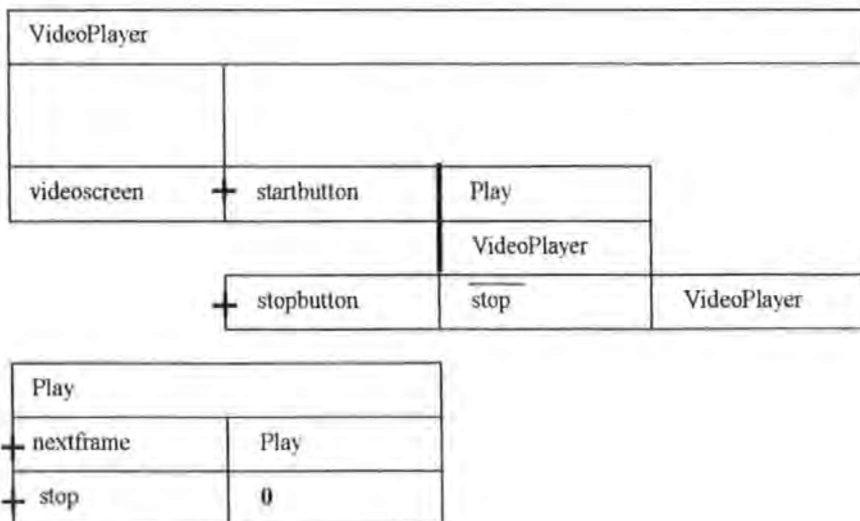


Figure 4.2: The video player in the graphical notation

In this example a "storyboard" box has been used for the initial *videoscreen* event to allow the required screen to be illustrated. All other boxes relate to simple events or invocations of other processes, so they have been compacted to just the label.

The important feature of this notation is that although it looks like a conventional storyboard albeit with the enforcement of some more rigorous layout conventions than most information designers would be used to, it is in fact simply CCS using a graphical, rather than textual symbol set. Thus storyboards produced in this way may be readily translated to CCS expressions. Indeed, if they are created on a computer using a suitable editor then the translation can be automatic. The CCS expressions can then be verified using standard procedures and tools.

4.17 Software tools

The notation described above works quite feasibly using paper. An extension is possibly needed to allow definitions to extend over multiple sheets of paper. This is achieved by leaving boxes open ended to the right to signify that the box continues on the next sheet. Tthis will usually occur only with the name box at the top of a definition. The continuation is likewise open ended to the left, with the name repeated to avoid the need for reference back to preceding pages. A continuing sequence of boxes is indicated by ellipsis to the right, with the continuation indicated by ellipsis to the left. All continuation should occur at the same height on the page as the continued boxes. An extended version of the video player example, with continuation onto another sheet, is shown in figure 4.3.

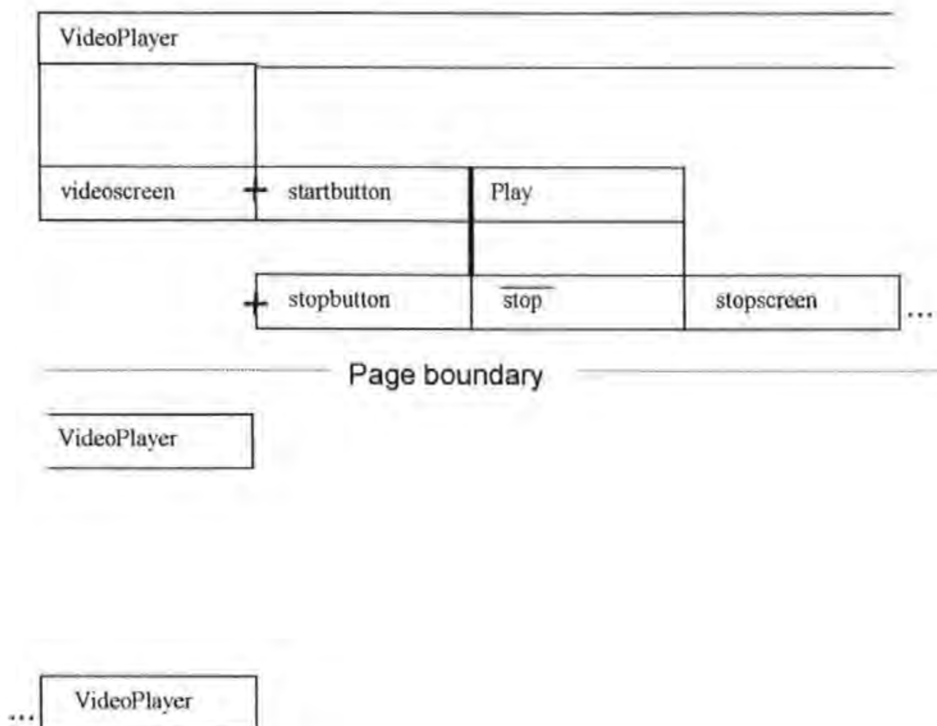


Figure 4.3: Page continuation

While with these extensions paper is a viable medium for the storyboard notation, much will be gained by generating it on a computer. Although, being graphically very simple, it can be generated using almost any 2D graphical editor, and some word processors, a purpose made editor would allow additional functions that will be valuable in producing a complete methodology based on the method.

The content of the storyboard will be in the form of image, movie, simulation and other content files. In the paper based notation this can only be indicated by an illustration in the storyboard box and possibly a written reference to the file. Using a computer based editor this written reference could be replaced by a hypertext link, which would allow the definition files output by the tool to include references to the correct file. It would then be

possible to produce further tools to be used later in the production process which would automatically compose the correct content files into the appropriate part of the sequence.

4.18 Analysing and proving designs

The process of verification is described here in outline. Firstly it should be noted that there is no question within the current state of the art of proving total, or even partial, correctness of a specification. What is required is to be able to demonstrate the presence of some important properties. In the case of our safety critical documentation systems discussed earlier these properties are the safety properties that have been defined along with the original system specifications. HML allows the construction of properties to express the satisfaction of conditions concerning the presence or absence of specified agents at particular times. There are several extensions to HML. The one that gives the minimum possible coverage of the analysis requirements here is THML^{+}, as described in [Fencott, 1996],which includes a linear time temporal logic (hence the prefix T, for Temporal). The syntax of THML^{+} is defined here in BNF.

$$P ::= tt \mid \neg P \mid P \wedge P \mid [K]P \mid \{t\}P \mid \mathbf{G}P \mid P\mathcal{U}P$$

P is a property, tt signifies *true*, $\neg$ signifies negation, $\wedge$ conjunction, $[K]P$ signifies that the occurrence of an action in the set K of necessity leads to condition P, $\{t\}P$ signifies that before some instant t P may be

satisfied, GP signifies that P will always be satisfied and PUP signifies that P is satisfied until Q is true. In addition to these definitions there are derived operators and results, as follows.

$$ff \stackrel{def}{=} \neg tt$$
$$\langle a \rangle P \stackrel{def}{=} \neg [a] \neg P$$
$$\mathbf{F}P \stackrel{def}{=} \neg \mathbf{G} \neg P$$
$$\langle a \rangle tt$$
$$[a] ff$$

These signify *false*, that after a it is possible to satisfy P, that at some time P will be satisfied, all agents which can accept a and all agents which cannot accept a respectively. The formal semantics and derivation of these results is given in the cited work.

Given a set of conditions in HML the task is to prove or disprove these conditions. This is done by finding agents within the specification which are equivalent to those within the conditions and demonstrating that they occur only within the temporal bounds expressed in the HML. It should be noted that the goal is to find equivalence, not identity. Several possible levels of equivalence exist ranging from weak, or observational, equivalence, which requires only the externally observable behaviour of the agents to be equivalent to strong equivalence, which requires internal and external behaviour to be the same. For our purposes observational equivalence is sufficient. A strengthened form of observational equivalence, observational congruence, forms the basis of a set of algebraic laws which

allow the manipulation of CCS equations (or, more accurately "observational congruence-ions"). These laws allow the transformation of CCS formulae into equivalent, or observationally congruent, forms. By manipulation of the specifications using the laws we can construct mathematical proofs where the propositions are the conditions that must be demonstrated.

Obviously, such a task is daunting, even for an accomplished and patient mathematician. Luckily, proof automation is a rapidly advancing technology and tools such as Jape [Bomat, Sufrin 1994], the B-Tool [Bieber, 1996] and, for CCS, the Concurrency Work Bench [Moller, 1992] are available. The complexity of CCS and the associated laws is much smaller than for other process algebras, so satisfactory proof assistance should be relatively straightforward.

At the end of this process it will have been demonstrated that the specified safety conditions are met, at least as far as external observation is concemed. It is not possible to verify the sequence of intemal events in this way, but as they are not observable they do not affect the users perception of the system. Of course, none of this guarantees the correctness of the original specifications, or that the specified set of safety conditions is correct or complete, but it should ensure that so long as the implementation is an accurate refinement of the specification it will not violate any of the safety conditions which have been specified.

4.19 Structures produced using process algebras

It is worth observing at this stage that the structures that will be produced using this method are quite unlike conventional hypertext systems. Hypertext uses a simple link, and thus is structurally a simple directed graph, with no control over structure of such systems are prone to produce structures known colloquially amongst programmers as "spaghetti". The simple link is, in terms of control structures, the precise equivalent of the old "goto" command in assembly code and simple programming languages such as BASIC (and, incidentally, most scripting languages). Programmers using these languages were often accused of producing "spaghetti code", but the structure produced were much simpler than that in hypermedia systems, which are likely to produce results more akin to a pasta factory.

Being an essentially structured method, the method proposed here gives a much more structured product. In particular, invocation of new pages of information is done in an environment which retains the original context. In programming terms the invocation is a "call and return" rather than a "goto".

4.20 How the method is used

Using the proposed method the steps in designing a multimedia documentation system are as follows.

The starting point for the design is a statement of requirements or requirements specification. This needs to detail the purpose of the system,

the overall structure, the data sources and the safety conditions, the properties that must be preserved in any implementation of the system. These specifications are non-formal in that they are prose specifications, not mathematically framed requirements.

The next stage is to design the top-level structure of the system, defining it as a system of processes using the graphical or textual process algebra notation. This stage will require the first major design decisions to be made as to the operation system – for instance in a multi-purpose system whether it is designed as a moded or modeless system. The output of this stage is a set of top-level process specifications. These specifications go through a process of successive refinement, detailing the internal structure of the processes in terms of other processes and ultimately single events. At each stage of refinement any safety conditions relevant to that level of abstraction must be framed formally and the specifications verified for consistency and for presence of the safety conditions using the appropriate analysis logic.

At the end of this process the design is complete. The design will provide a set of storyboards for the artists and designers to use, and can be translated into a formal specification of the sequencing of those events, a sequence which has already been verified as meeting the initially specified safety requirements. This sequence needs to be translated into a program that will sequence the images, graphics, models and sounds generated by the designers. The most likely way of doing this is using a structured scripting language such as HyTime or JavaScript or a programming

language such as C, ClassiC [Newman, Payne, 1994] or Java. Note that the method produces a structured program style, call and return invocation of processes, rather than a "goto" style control transfer as is typical of hypertext, so conventional hypertext scripting languages such as HTML or Director Script are not suitable for this purpose. Using a suitable scripting language the translation from specification to program is straightforward and could be made automatic.

4.21 Separation of structure and content

One of the important points to note about the discussion on the use of this method is that, by applying the method, we have separated out the dynamic behaviour and structure of the system from the content. This separation occurs when the "storyboard" notation, which contains a formal definition of the structure and an informal indication of the content, is translated into the textual process algebra notation, which contains only structure and dynamic behaviour. This separation of concerns should achieve some of the goals of increasing authoring productivity by allowing the use of libraries of pre-verified process structures to be used templates into which the content itself can be slotted. Such a system could be used to cater for detail variations in documentation without the need for rewriting he whole of the specification. Future developments of the notation could include some more formal definition of the content, such as indicators for frame and control style and positions, and allow the production process from the initial specifications to be more completely automated.

4.22 An example

This section introduces a worked example of the operation of the method. The example is taken from a motor car maintenance manual (both car and manual will remain anonymous) and is interesting because it demonstrates the complexities that can quickly arise in the design of multimedia documentation and also that conventional manuals can also contain procedural bugs. The part of the maintenance procedure we are concerned with is the periodic changing of the camshaft drive belt. In the car concerned this procedure differs depending on whether the engine is in or out of the car. If the engine is out of the car, say for a general overhaul, then all that needs to be done is to remove the cam belt covers and swap the belt. If the engine is still in the car then the procedure is more complex since the belt passes around one of the engine mountings. Since the belt is continuous it can only be changed if the engine mounting is first removed. Removing the engine mounting involves partially dismantling the front suspension to gain access, and that in turn requires the jacking up of both car and engine.

This state of affairs raises some fairly basic system design issues. On the one hand, the two procedures can easily be conducted with a purpose made sequences of pages, but this would require unnecessary duplication of the data, and along with that would come unnecessary duplication of authoring resources. Instead we need to design two separate processes, for engine overhaul and service, which both make use of sequences which are common. As a safety condition, we need to ensure that no attempt is

made to remove the mounting if it is supporting the engine. For the purposes of designing the documentation we make the maybe rash assumption that being shown the page describing that part of the procedure is the same thing as carrying out the procedure itself (in really critical applications the documentation system would require the engineer to confirm each procedure as it was carried out). Several other safety conditions for this procedure are obviously required, such as ensuring that the engine mounts are not removed before the engine has been supported, or ensuring that if the front suspension is dismantled then it is remantled, but for the purposes of this example the one condition will suffice.

One design aim will be to use as much common content as possible. For this reason we will specify a single agent for the actual change of cam belt, as shown in Figure 4.4.

ChangeCamBelt				
removecovers	securepulleys	changebelt	freepulleys	replacecovers

Figure 4.4: The change cam belt agent

This is a simple sequence of five screens illustrating the procedures for removing the protective covers, securing the belt pulleys to maintain their respective alignment, swapping the belt, removing the restraints on the pulleys and replacing the covers. This agent can be used, so long as the engine mounting has been removed. There are several ways of

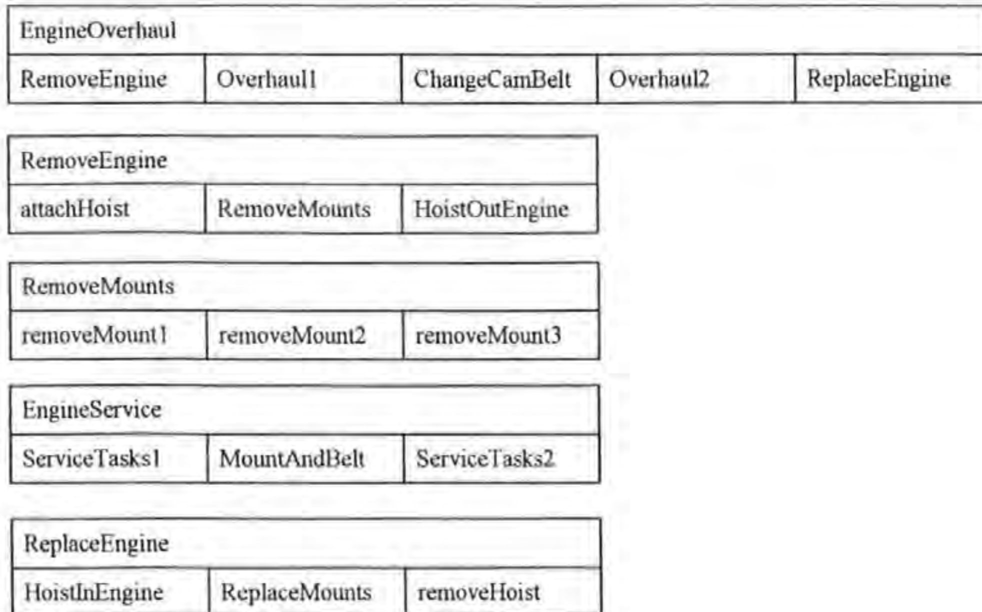
ensuring that this is done. One is to use the choice operator as shown in figure 4.5.

CheckAndChange		
engineInCar?	no	ChangeCamBelt
	yes	MountAndBelt

MountAndBelt		
removeMount1	ChangeCamBelt	ReplaceMount1

Figure 4.5: Asking the user to select the procedure

This displays a prompt screen, and depending on which selection the user makes selects either the raw ChangeCamBelt agent, or else one prefixed by an agent illustrating the procedure for removal of the engine mount and followed by one illustrating its replacement. Note that the responses to the prompt have hidden names, since the names "yes" and "no" are likely to be well used elsewhere. In operational terms this is likely not to be the optimum solution, since it involves a user intervention at a critical stage of the process. Discussions with users have shown a clear preference of task-orientated documentation, so a preferable option would be include the appropriate agent into the agents for the engine overhaul and the engine service. These are shown in Figure 4.6.



EngineOverhaul				
RemoveEngine	Overhaul1	ChangeCamBelt	Overhaul2	ReplaceEngine

RemoveEngine		
attachHoist	RemoveMounts	HoistOutEngine

RemoveMounts		
removeMount1	removeMount2	removeMount3

EngineService		
ServiceTasks1	MountAndBelt	ServiceTasks2

ReplaceEngine		
HoistInEngine	ReplaceMounts	removeHoist

Figure 4.6: Engine overhaul and engine service task based procedures

These procedures make use of the previously defined MountAndBelt agent. Some dummy agents, Overhaul1, Overhaul2, ServiceTasks1 and ServiceTasks2 have been introduced to represent the details of these tasks not relevant to this discussion.

Since these storyboards are simply a graphical version of CCS, they can be simply transformed into CCS, to give the following definition equations.

$$ChangeCamBelt \stackrel{def}{=} removecovers.securepulleys.changebelt.freepulleys.replacecovers$$

$$CheckAndChange \stackrel{def}{=} engineInCar?.(no.ChangeCamBelt + yes.MountAndBelt) \setminus \{yes, no\}$$

$$MountAndBelt \stackrel{def}{=} removeMount1.ChangeCamBelt.replaceMount1$$

$$EngineOverhaul \stackrel{def}{=} RemoveEngineOverhaul1.ChangeCamBelt.Overhaul2.ReplaceEngine$$

$$RemoveEngine \stackrel{def}{=} attachHoist.RemoveMountsHoistOutEngine$$

$$RemoveMounts \stackrel{def}{=} removeMount1.removeMount2.removeMount3$$

$$EngineService \stackrel{def}{=} ServiceTasks1.MountAndBelt.ServiceTasks2$$

$$ReplaceEngine \stackrel{def}{=} HoistInEngine.ReplaceMounts.removeHoist$$

Similarly, our safety condition, that no attempt is made to remove the mounting if it is supporting the engine, can be translated into HML. We require to say that all agents including the event *removeMount1* will occur after the event *takeStrain*, which ensures that the weight of the engine is supported. This is essentially a statement based on the state of the system. As noted earlier, these conditions are framed much more easily using predicate logic, using an event based logic the framing of the condition becomes quite difficult. This is particularly the case with HML, which deals with future potential, rather than past traces, as does, for instance, the traces semantics of CSP. One way of capturing the state is to observe that the hoist can only be removed if it has been attached. We can only be sure of this if we separate the agents in the system which occur after *attachHoist* and ensure that only they contain *removeHoist*. This property, of the system as a whole, is expressed as follows.

$$HA \stackrel{def}{=} \langle attachHoist \rangle \mathbf{F} \langle removeHoist \rangle tt$$

HA signifies that acceptance of *attachHoist* leads to a state which at sometime will be satisfied by an agent accepting *removeHoist*. Thus there are no *attachHoists* not matched by a *removeHoist* – this is an important correctness property in its own right. The property $\neg F \langle removeHoist \rangle tt$ denotes the set of agents from which an agent accepting *removeHoist* will not occur at some time, so agents satisfying *HA* exclude those which will not accept *removeHoist* at some time.

So long as HA is satisfied for the system as a whole we can be sure that acceptance of *removeHoist* implies a previous acceptance of *attachHoist.*, so long as we don't meet an *attachHoist* along the way This property can be expressed as shown below.

$$HB \stackrel{def}{=} [attachHoist] ff \mathcal{U} \langle removeHoist \rangle tt$$

There will be no state accepting *attachHoist* until *removeHoist.* The last part of the condition is to specify that any state accepting *removeMount1, 2* or *3* must lead to a state satisfying HB. This can be framed as follows.

$$HC \stackrel{def}{=} [removeMount1, removeMount2, \mathrm{Re}\, moveMount3] HB$$

Verification of these properties can be achieved in a number of ways, including algebraic manipulation and exhaustive specification animation. In a system of this size the latter is simpler. If we consider the *EngineOverhaul* agent, this can be expanded to the following

$$\begin{array}{l} EngineOverhaul \stackrel{def}{=} \\ attachHoist. \\ removeMount1.removeMount2.removeMount3 \\ .Overhaul1.removecovers.securepulleys.changebelt. \\ freepulleys.replacecovers.Overhaul2. \\ HoistInEngine.\mathrm{Re}\,placeMounts.removeHoist.\mathbf{0} \end{array}$$

Note the null agent *0* has been appended to make this a complete system specification. This trivially satisfies HA, since the agent starts with *attachHoist* and finishes with *removeHoist*. The set denoted by

$\neg F(removeHoist)t$ is simply $\{0\}$, which clearly satisfies the condition. The only agent following *removeMount1, 2* or *3* is

Overhaul 1.*remove* cov*ers*.sec*urepulleys*.*changebelt*.
freepulleys.*replace* cov*ers*.*Overhaul* 2.
HoistInEngine.Re*placeMounts*.*removeHoist*.0

Which clearly satisfies *HB*.

4.23 Comparison with Eventor

The authoring tool *Eventor* [Eun et. al., 1994] was discussed in chapter three. Eventor is uniquely related to the method put forward here in that it too is based around the CCS process algebra. For this reason, the detailed differences between the two are discussed here.

One fundamental difference is the view of the system structure presented to the user. Eventor seeks to abstract away from the underlying CCS structure by presenting three different views of the presentation to the author, a temporal synchronizer, a spatial synchronizer and a user interaction builder. By contrast, the present work adopts a single view, the storyboard, that is both directly, and explicitly, derived from the CCS and also is similar to existing storyboard notations used in common practice. The illustrations in the storyboard contain the spatial and content specification.

Eventor seeks to conform to an object-based view of the system, by identifying as the basic building blocks for the system "basic objects" that

correspond to CCS agents rather than events. The objects have internal structure, which specifies the communication and synchronisation between objects but not the sequence of presentation within the object. To cater for different types of behaviour it has been necessary to define different types of object and a separate type of composite object to compose defined agents together. The storyboard method identifies events as the basic building block of the system. These events include presentation, interaction and synchronisation and the author is expected to specify them explicitly using the storyboard notation. The CCS agents produced include both presentation and synchronisation specifications. Since a CCS agent is simply a collection of events there is no need for a separate composition object to compose other objects together, agent definitions compose events and other agents freely. This is one of the advantages of CCS over, for instance, CSP that requires different composition operators for agents and events and does not allow hem to be freely composed together. The specifications produced by Eventor are very "CSP like" in that they include many agent definitions which have the sole purpose of composing agents and events. For instance, in [Eun, No et al, 1994] an example is given of a video player agent.

$teacherVideo = SCAI1?.PlayVideo$
$PlayVideo = s1!.PlayVideo1$
$PlayVideo1 = s2!.PlayVideo2$
$PlayVideo2 = s3!.PlayVideo3$
$PlayVideo3 = s4!.PlayVideo4$
$PlayVideo4 = s5!.endVideo$
$endVideo = endVideo!.teacherVideo$

This example is presented in its original form from the cited work. The CCS notation used is different to that used in this thesis, in that a simple equality symbol is used rather than the "definition equality" used here, and in that a synchronisation event is represented by an $e?,e!$ pair, rather than the notation used here. The normal convention of starting event names with a lower case and agent names with an upper case has not been used, and there is some confusion between events and agents. The final two lines of this specification are incorrect CCS, since an agent is used as a synchronisation event, a semantic and syntactic impossibility. The example given is presented again here, corrected and translated to the version of the CCS notation used in this thesis.

$$TeacherVideo \overset{def}{=} \overline{scai1}.PlayVideo$$
$$PlayVideo \overset{def}{=} s1.PlayVideo1$$
$$PlayVideo1 \overset{def}{=} s2.PlayVideo2$$
$$PlayVideo2 \overset{def}{=} s3.PlayVideo3$$
$$PlayVideo3 \overset{def}{=} s4.PlayVideo4$$
$$PlayVideo4 \overset{def}{=} s5.EndVideo$$
$$EndVideo \overset{def}{=} endVideo.TeacherVideo$$

By substitution this can be translated to a single agent definition.

$$TeacherVideo \overset{def}{=} \overline{scai1}.s1.s2.s3.s4.s5.endvideo.TeacherVideo$$

This substitution makes the content of the agent much clearer. It may be noted that the agent includes only the synchronisation events, with no

presentation information and that after the last video clip there is a sequence of two synchronisation events, $s5$ and $endvideo$. It is not known whether this was the intention of the authors. Insertion of events to represent the playing of the clips gives the following.

$$TeacherVideo \stackrel{def}{=} \overline{scai}1.s1.clip1.s2.clip2.s3.clip3.s4.clip4.s5.clip5.endvideo.TeacherVideo$$

In the example given the purpose of the synchronisation events $s1$-$s5$ is to synchronise the playing of the sound tracks to the video clips. In the Eventor model, where objects are represented by agents, these must be separate agents, each specified using a specification of several lines of definitions, similar to that for $TeacherVideo$. If CCS is used in the way suggested in this thesis the sound clips are represented by events in the same way as the video clips, and can simply be composed in parallel with the corresponding video clip, as shown below.

$$TeacherVideo \stackrel{def}{=} \overline{scai}1.(clip1 \mid sound1).(clip2 \mid sound2).(clip3 \mid sound3).(clip4 \mid sound4).(clip5 \mid sound5) .endvideo.TeacherVideo$$

This can be represented graphically using the storyboard notation, which expresses the relative sequencing of the sound and video clips clearly. The content boxes have been omitted, since there is no indication of the content of the video clips in the cited work.

TeacherVideo							
$\overline{\text{scai}}$	clip1	clip2	clip3	clip4	clip5	endvideo	TeacherVideo
	sound1	sound2	cound3	sound4	sound5		

Figure 4.7: Storyboard notation for Eventor example

The authors of Eventor identify the use of formal specification as one of the advantages of their system and suggest that formal specification may allow verification of the correctness of the syntax of the system. The use of formal specification is one of the requirements of the storyboard method, in order to allow the semantic verification of safety critical documentation systems. The possibility of semantic verification in Eventor is limited because the agents do not include any content events, and therefore it is not possible to reason about their sequence of presentation. Eventor is therefore not suitable as a specification method for safety critical systems.

4.24 Conclusions

The method proposed for design of multimedia systems is underpinned by the theory that has been developed to allow the rigorous design of safety critical real time software systems. It relies on a body of theory that has proved to be quite accessible, at least in underlying concepts, to those with experience in the programming disciplines. The semi-graphical notation that has been developed provides a means both to link in the content of the system (the images, models, sound and movies) and to provide a way of defining the sequence of presentation in a way that

is similar to traditional storyboard techniques. A system specification developed using this notation can be translated into a textual process algebra, and is then susceptible to analysis and verification using the techniques of that algebra. This specification method can form the basis of a complete methodology allowing the development of large, multimedia technical documentation systems, a need that has been observed and commented upon frequently within technical documentation operations within industry.

Chapter 5: Implementing high reliability multimedia systems

5.1 Introduction

The first part of this thesis has shown how process algebras may form the basis of a method for the formal design of hypermedia systems for use in technical documentation applications, and indeed, any applications where reliability and correctness are at a premium. Formal design methods provide a means of producing a specification for a product that has been verified against some identified requirements. In the case of this method the specifications can be verified against a set of *safety properties*, that is conditions, generally sequences of presentation, that must be maintained if the processes that the documentation is guiding are to be correctly performed.

However, a verified specification is of little use if the final realisation of the system does not preserve the properties established in the specification. In order to be sure of this the implementation process must maintain the function defined all the way down to the machine code running on the computer system. The process of implementation is a process of

refinement. Specifications that operate at one level of abstraction are refined into implementations at a lower level of abstraction. This implementation in turn provides the specification for the next step of the refinement process until ultimately the output is an executable program (which is a specification of the sequence of states that the hardware must go through to provide the specified sequence of sounds, images and interactions for the user). To maintain complete confidence in the refinement process requires that each link in the process, from specification to executable code, must be susceptible to verification against the next higher level of specification.

There will be some parts of this process where it is simply not possible to provide the next link in the chain of verifiability. For instance, there is no way that the formal specification can be verified against the initial prose specifications, since English (or any other human language) does not have formally specified semantics. Even were it to, there can be little confidence that such semantics would actually correspond to the intentions of those who wrote the specification. At this stage in the chain we have to rely on informal processes such as review and "walk through" of the specification by domain experts.

Similarly, it is impossible to complete the chain at the other end of the refinement sequence. Unfortunately, there is little practical choice but to accept the quality of commercial language implementations. While few have been formally verified most have at least been extensively tested, although this does not guarantee correctness.

A third area where we cannot provide any formal level of confidence is that of the content itself. There can be no assurance as to the correctness of the message conveyed by the images or models in the documentation. There is little hope of including rigorous semiotic analysis into the method. Ensuring the appropriateness and correctness of the content will have to rely on sound design practice and established quality assurance methods.

Even though the links cannot be made at the end of the chains, there is value in maintaining the sequence of verification for the other processes involved. Errors can easily be introduced in the refinement process and the process of formal verification helps detect those that have. Another means of preventing introduction of human errors is the automation, so far as is possible, of the refinement process.

This chapter investigates the possible processes of refinement for hypermedia systems whose specifications have been produced using the method introduced in the first part of the work.

5.2 The form of the specification

The first stages of the design method are the storyboard production, its translation to CCS and the formal verification structure. The output from these stages is a formal definition of the dynamic structure of the system, in the form of a CCS specification, and an indicative definition of the content, in the form of the illustrations in the storyboard frames. The refinement task

consists of two separate jobs: generating the content and animating the dynamic structure that contains that content.

Content generation may use a number of methods depending on the nature of the medium. The precise methods of content creation are outside the scope of this work. It will be sufficient to assume that the contents are generated and stored in a heterogeneous series of content files. One assumption that has been made consistently throughout this work is that the contents are *static*, that is that they contain no behavioural information beyond display of information for a period of time. All information on the dynamic behaviour of the system must be contained within the dynamic structure definition contained in the CCS specifications. To assume otherwise prejudices the ability to make any rigorous analysis of the dynamics of the system since dynamic behaviour will exist which is not described by the CCS formulae. This requirement does place a limitation on the formats in which the information may be held, or at least how they are used. Formats which contain dynamic information and links, such as HTML, may be used so long as no dynamic information is included. In essence the situation is similar to that in a link or database service based hypermedia system, the dynamic information must be abstracted away from the content. Objects such as video clips may be represented as a CCS agent, since they are simply a sequence of frames with no internal choice or concurrency. Their behaviour is predictable and they cannot interact with other objects except by playing through their full sequence. Those that do need to interact with other agents, as, for instance, the video

player example given in Chapter Four, must be provided with a complete CCS specification defining their behaviour.

The dynamic information needs to be constructed using a notation that has a good semantic match with CCS. CCS, in common with all process algebras, produces a program like structure, in which actions map to instructions, agents to procedures, choice to "if" instructions and parallelism to program forks. Thus the most natural structure to map CCS into is a programming language, and moreover one which contains these elements. There are very many different programming languages and scripting languages that could be suitable. The target language should also have the characteristics of modularity and data and control abstraction that would be necessary for the implementation of large-scale systems by teams of programmers. Recently the trend has been for such languages to conform to the *object-oriented* programming paradigm.

As is the case for much technical terminology, the term *object oriented* has been sufficiently abused by marketing executives and journalists to render its meaning ambiguous. Object oriented languages are usually identified by possession of a set of characteristics, rather than by adherence to a hard definition. Such a set of characteristics is identified in [Bal, Grune, 1993] as the following:

Encapsulation of the state of the program into *objects*. An object contains data and provides operations for accessing these data. The operations are the *interface* to the object for users of the object

Use of the principle of *data encapsulation* to establish a firewall between the user of an object and the code implementing it, thus achieving information hiding.

The provision of *inheritance*, to allow different kinds of objects to be built hierarchically, with the most general at the top and the more specific ones at the bottom.

The use of *dynamic binding*. Since code is encapsulated with data in objects, and objects will be bound at run time, the selection of code to be run will be made at run time.

The use of *type polymorphism*, so that a procedure (or method) may accept parameters of different data types, so that a formal parameter can correspond to actual parameters of different types in different calls.

Several of these characteristics will be very helpful for the implementation of hypermedia systems. By their nature, hypermedia systems will contain many different types of display object. Polymorphism will allow them to be handled in a common, consistent way.

Data encapsulation, and the information hiding that comes with it, provides the means for large teams of programmers to co-operate successfully. The interfaces between their individual pieces of code are tightly defined by the class definitions that define the objects and information hiding guarantees that the internal state of their objects is not vulnerable to unintentional modification by some other programmer.

For this application we are also looking for languages which are amenable to formal analysis and verification, in order to allow the chain of verification to be completed from the initial storyboard specification to the final executable code. From this point of view, object oriented languages are simply imperative languages, with the same sequential, instruction ordered semantics. Thus such languages, if their semantics are formally specified, are amenable to verification using established methods such as Hoare logics [Hoare, 1969] or weakest precondition calculus [Dijkstra,1976]

The object based structures of these languages means that the order of development of a proof becomes somewhat different from that for non-object-oriented languages, although the underlying principles remain the same. Meyer [Meyer, 1993] has introduced a specification and verification method for object-oriented languages called "design by contract". Here the weakest preconditions required for use of each method and the post-conditions after execution for each object are included as assertions within the definition of the object. The object can be internally verified to comply with the conditions using conventional proof methods. Users of the object can now adopt these assertions as defining the behaviour of the object, and can use them to produce the pre and post-conditions of any call of any method of the object.

Thus, object-oriented languages, as a class, would appear to be a good implementation target for the design method. As stated above, the target language will also be required to support concurrency. It is therefore

likely to belong to a class of languages called Concurrent Object Oriented Languages (COOLs)

5.3 Introduction to COOLs.

It has been observed by Bertrand Meyer, the designer of Eiffel, that there is an obvious match between many of the properties associated with concurrent programming constructs and those supporting object orientation. In particular both support local variables, persistent data, encapsulated behaviour, restrictions on exchange of information and a communication mechanism often modelled on some form of message passing.[Meyer, 1993] Concurrent Object Oriented languages (COOLs) exploit these similarities to create programming systems which support both concurrency and object orientation in an integrated way so that the facilities supporting object orientation, such as type inheritance, data abstraction and polymorphism are also available to support concurrency. Such languages include Eiffel// [Caromel, 1990], POOL [America, 1987], ACT++ [Kafura, Lee, 1990], Java [Sun, 1995][Sun 1996], and ClassiC[Newman, Payne 1994][Newman 1995]. There are also languages in which the concurrent and object oriented extensions have been made in an orthogonal way, so that the two sets of constructs are separate. This group includes Concurrent C++ [Gehani, Roome, 1988] and Ada95[[Ada9X 1992a][Ada9X1992b]. Such languages are not only syntactically larger but they lack some of the expressive power possible in the true COOL.

Minimising the amount of additional syntax has been an important consideration in the design of COOLs. Meyer, in [Meyer, 1993] cites this reason as a reason for not including conventional synchronised inter process communications methods within a COOL. In COOLs the process of a conventional concurrent language becomes simply an active instance of an object. Inter process communication is performed simply by calling the methods of that object. If such calls are synchronised it is necessary to include some facility for a selective wait or an exception mechanism to provide the necessary non-determinacy. It is argued that such an extension will clutter the syntax of the language and negate some of the advantages of the integration of concurrent and object oriented programming constructs. Caromel argues that asynchronous communication relieves synchronisation dependencies between classes, allowing them to be self contained modules [Caromel, D 1993]. However in real time systems, control of synchronisation is an important issue, as is the analysis of timeliness of communication and susceptibility to deadlock and livelock. These analyses are simplified using a synchronous communications model, which is amenable to the methods established by Hoare using CSP [Hoare, 1978].

Concurrent real time systems require structured programming methods in two domains. The first domain is that of procedural structure as with non-concurrent systems. In this domain object-oriented structure has become a favoured paradigm and has been reflected in the development of object oriented design methods specifically for real time systems.

The second domain of structure is that of concurrency. Although there is a school of thought that sticks to the certainties and predictability of the cyclic executive [Burns, Welling, 1990], many practitioners in real time systems favour a system structured as concurrent co-operating processes. Hoare [Hoare, 1972] has demonstrated how such systems are amenable to formal analysis and can thus deliver the same degree of predictability as the cyclic executive with the added advantage of improved program structure, clarity and maintainability.

COOLs use method calls as the vehicle for inter process communication. The control and synchronisation of access to the methods varies. In one model access to methods is controlled by the internal state of the object to which they belong, each method is identified with named states in which access is permitted. Typical of this approach are actor languages such as ACT++. Another model provides completely asynchronous method calls, with calls buffered until the object can handle them and results buffered until used by the calling process, as is the case in Eiffel//. It is claimed that such a scheme simplifies programming and eliminates unnecessary synchronisation and serialisation. It does, however, make predictable synchronisation difficult to achieve and also makes program verification significantly more complex.

Chapter 6: Translating CCS specifications to COOLs

CCS specifications produced using the subset of the notation which has been used here require the following constructs to be translated into a program structure. These are:

- The basic entities, the events.
- Compositions of events, the agents.
- Agent definitions.
- Prefix, the '.' Operator.
- Choice, the '+' operator.
- Association, the '(' and ')' operators.
- The parallel composition operator '|'.

6.1.1 Events

Events can be subdivided into several categories. These are: *display events*, which cause the display (or replay) of some kind of data; *user events*, which create some kind of user control; *input events* which respond to user actions and *synchronisation events* which cause synchronisation between agents.

Display events.

A display event is mapped to a call of the appropriate display method of an instance of a specialised class that displays the appropriate type of object, using the appropriate data. Such class definitions will form part of the environment in which the translation process is undertaken, and will map into the corresponding system calls to load and present the data. One advantage of object-oriented program construction in this application is that a single class definition may be made to serve a number of different media types, using the properties of polymorphism and type inheritance. These properties also make it possible to include new data types into the system without the need to rewrite all of the class definitions. The new types are handled as extensions of the old types.

User events and input events

User events come in the form of posting interaction controls that will at a later time be responded to and will cause input events. Because the two are so closely associated they are dealt with together here. Like display events, user events will map to a call of a method in an appropriately designed control object, causing the required interactive control to appear on the screen. Generally user input in COOLs is dealt with by inter-process communication from an imaginary process representing the outside world. If this mechanism is used then the user input will come in the form of an inter-process communication, for most COOLs a method call.

Synchronisation events

Synchronisation events will use the synchronisation method provided in the COOL. If the COOL uses synchronous communication this will simply be an inter-process communication action.

6.1.2 Prefix, the '.' Operator.

The '.' Operator represents sequential composition. In a sequential programming language such as a COOL this is represented simply by using the instruction sequence of the language.

6.1.3 Agents.

Agents need to be classified as either belonging to the set of agents that must be capable of sustaining an independent thread of activity or those which are simply convenient groupings of actions. The former will require a process, in COOL terms an active object while the latter may be implemented simply using a procedure, while the. A tidier solution would be to use an object (i.e. a class definition) to represent both, with the difference that the agent which requires an independent thread of activity is an active object as opposed to a passive one.

6.1.4 Agent definitions.

If agents are represented as method and class definitions the association of the agent name with its definition is automatic. Where name hiding is required the appropriate scope rules of the COOL may be used to ensure that the name is invisible outside the object.

6.1.5 Choice, the '+' operator.

The provision of the choice operator is more complex. Some COOLs provide a direct equivalent to the CCS choice operator, which is modelled on the Dijkstra multi-armed "if" statement [Dijkstra, 1976] and for these the translation will be directly to that statement. For those that do not, a majority of COOLs, some more complex mapping will need to be performed.

6.1.6 Association, the '(' and ')' operators.

The parenthesis operators provide explicit control of the association of the composition and choice operators. This control is provided by the block structure in a block structured programming language, which most COOLs are.

6.1.7 The parallel composition operator '|'.

Finally we consider the parallel composition operator. We can consider the production of a parallel composition operator to be a process fork. Those COOLs that provide an explicit process fork will be able to provide a direct mapping. Once again, those languages that do not provide this facility will require a more complex translation.

6.2 The ClassiC language.

The author has developed a COOL which has the characteristics which identified above as necessary for the implementation of hypermedia systems using the method. This language is called ClassiC (an

abbreviation of Class integrated Concurrency). ClassiC has adopted a synchronous method call scheme. The semantics of the method call are very similar to those for an Ada rendezvous and as such analytical and verification techniques for it are well understood, essentially being an extension to the semantics of synchronous message passing as discussed by Hoare and others.

6.2.1 Object oriented concurrency.

ClassiC builds its model of concurrency around the idea that a process is simply an object which has a strand of processing associated with it, in terms that have been used elsewhere, an active object. This is differentiated from passive or inactive objects, which rely on activation of one of their methods for any activity. This is in line with the design approach used in most COOLs, but differs from that used in Meyer's concurrent extension to Eiffel, which adds the abstraction of a *processor*, and ACT++, which is based around the actor model of concurrency. While each approach has its proponents, the use of the process model does have some key advantages.

1. The process abstraction is familiar and well understood by programmers of concurrent systems and many design methods are based around it.
2. The process model fits naturally with formal methods based on process algebras, including CSP, CCS and LOTOS.

3. The process model fits neatly with process based operating systems and schedulers. This is particularly important in the field of real-time systems, where the majority of work on schedulability has been done using process models.

A process in the ClassiC model has exactly the same properties as objects (defined by classes) and in addition possesses an independent strand of processing. The ideas of data abstraction, encapsulation and inheritance which are associated with a class in C++ are also properties of a process in ClassiC.

The unification of concurrency and program structure within the object-oriented model of structure, the class, is the definitive feature of all COOLs, including ClassiC. It brings with it a number of advantages over the alternative approach of separation of the concerns of concurrency and program structure. These can be summarised as follows:

1. The amount of syntax in the language is smaller, since one set of syntactic constructs supports both processes and classes.
2. The programmer's conceptual model is simpler. As discussed in the introduction, many of the concerns of processes and objects are the same. When there are two different sets of constructs the programmer is required to assimilate two similar, but different, entities.
3. The expressive power of classes is also usable for processes. The facilities available for classes, in particular inheritance, simple instantiation and initialisation, and interface definition are also available

for a process, resulting in a much richer process model than is normal in a concurrent language.

6.2.2 Inter process communication.

An object is defined by the operations that can be performed on it. Thus the definition of an active object, from the point of view of other objects, must also be in terms of the operations that can be performed on it. From the standpoint of C++ the operations are defined by the members of a class.

In the concurrent view of the world, objects are processes,and the operations and interactions between them are defined by the inter-process communication. If we are to adopt a unified object based model for both concurrent and non-concurrent objects then the model of inter-process communication must match the way that operations are modelled for non-concurrent objects, that is classes.

The conclusion drawn from this is that inter-process communication must be defined in terms of members of a class, usually functions. This leads to a picture of the inter-process communication method being the provision by a process of a set of functions which can be called by other processes, with suitable arrangements being made to ensure mutual exclusion between the two processes while that function is called.

This is in fact very close to the inter-process communication model provided by Ada, the Ada rendezvous, although simplified. The Ada rendezvous implies synchronisation at two points in the rendezvous, having

a body of code inside the accept statement. The ClassiC *accept* is a simple statement, and implies that the process is blocked at that point for the duration of the execution of the entry by the other process involved in the rendezvous. Actions taken as a consequence of the rendezvous must be coded around the accept instruction. Whereas Ada provides the rendezvous as a specific facility for inter-process communication, separated from the constructs used for data abstraction (the package) and inheritance (the tagged record) in the case of ClassiC all these concerns are bundled together, and the rendezvous fits naturally into the existing encapsulation and abstraction mechanisms.

Given the blocking nature of the inter-process communication mechanism it is necessary to have some means of either testing for readiness to rendezvous or a selective wait mechanism. As the former leads to awkward coding and encouragement of polling the latter has been used, based on Hoare's choice operator in CSP, as in Ada and occam 2 [INMOS, 1984]. The select statement is potentially a source of difficulty and inefficiency in implementation. We believe that the clarity and simplicity of programming offered by it outweigh this factor, and in any case ClassiC allows more efficient communication mechanisms between related processes. These can be used to create tightly coupled processes weth very efficient communication.

6.2.3 The ClassiC process.

The ClassiC process is simply an extension of the existing class construct. A class declaration may be annotated with a priority, which indicates that the class may have some active behaviour. The priority annotation serves the same purpose as the similar annotation in Modula-2: it signifies that mutual exclusion is guaranteed between processes accessing members of the class, that is instances of the class have monitor semantics.

There is more to a ClassiC process than a simple monitor, however. Instances of the class may also have associated with them their own thread of execution, or process. Such classes are called active classes. Those that don't are inactive classes. The means of association of the thread of execution with a class is unusual, and produces some particular characteristics which provide additional expressive power within the language.

The majority of COOLs imbue a class with activity by inheritance from a special active class, typically called *Thread* or *Process*. Once a class has inherited this active class it is active, and so will be all its descendants. The mechanism used within ClassiC is much more flexible, allowing active descendants of passive classes, multiple inheritance from active classes and even active and passive variants of the same class. The latter property is of use when activity is a secondary characteristic of an object, which modifies its temporal behaviour but leaves the logical specification the same. A typical example of the use of this is the introduction of buffering

into a system to selectively relax timing constraints. Using this characteristic the buffered and unbuffered variants of an object may be freely interchanged where this is appropriate.

A ClassiC class may have one or more members which are constructors, called at instantiation of objects of that class for initialisation. In an active class the constructors take on an added significance. After initialisation the constructor returns control to the process which called it by means of a *coretum* statement. *Coretum* causes a process fork, the parent process returning to the caller while the child continues execution of the constructor. The use of *coretum* , which is an original feature of ClassiC (*cobegin*, is a more commonly used construct) has a number of happy consequences. It fits easily into the structures of C++, placing the process fork clearly at the point where the parent process gives birth to the child process, the constructor function. The placing of the fork within the constructor also localises initialisation of process variables at the point of process instantiation.

As will be explained later, another effect of the use of the *coretum* method of causing a process fork, as opposed to inheritance from a special class which is used in some COOLs, is freedom from the "inheritance anomaly". Moreover provision of multiple constructors allows alternative process bodies to be provided for one class, or even for active and inactive variants of the same class. As far as the users of the class are concerned, they do not need to know whether the class is active or inactive, so long as

it performs the desired functions. Not only are data and control abstracted but processes are as well.

Once the active class has been defined, creation of the associated process is simply a matter of creating an instance of that class, in the same way as an instance of any other C++ class. No additional programming constructs are needed. The lifetime of the new process is the same as that of the instance of the class with which it is created. If the destructor for that instance is called the process will be terminated.

6.2.4 The ClassiC rendezvous.

The design of the inter-process communication mechanism within ClassiC has fallen out naturally from the combining of processes and classes. The means of communication between classes is by the calling of the member functions of one class by another and it is natural that communication between active processes should be achieved in the same way. However since concurrency is involved care must be taken to achieve the required mutual exclusion between the two processes.

In addition to the public, private and protected declarations that exist in C++, the ClassiC class definition may also contain entry declarations. These are members of the class, which are accessible to processes other than that which owns the class. Access to these members is controlled to ensure synchronisation and mutual exclusion between the two processes.

The mechanism used is similar to the Ada rendezvous. In Ada the shared procedure which forms the communication mechanism between tw

processes is called an *entry*. A process which accesses an entry will be blocked until the process which owns the entry executes an *accept* statement for that entry. Unlike the Ada *accept* the ClassiC version is a simple statement.

Once a process has executed an accept statement it is suspended until the corresponding entry has been completely executed, that is until the process using the entry has returned from it. The process executing the entry executes no code within the entry and so the complications that occur in Ada when an exception is raised within an entry do not occur. For the process using an entry, the entry has the normal semantics of a function call, except that its execution is synchronised with execution of the accept statement by the owner and thus exceptions may be raised in the normal way.

The entry itself is defined in precisely the same way as any other member of a class. While executing within that entry a process may have access to members of the class. Data hiding is associated with the class rather than the thread of execution. These rules are consistent with those for a normal (non-concurrent) class.

The ClassiC select statement is modelled on the C switch for reasons of syntactic consistency. The basic form of the statement is:

```
select guarded-statement
```

The guarded statement following the select consists of one or more arms each of which is a guarded statement with the form:

```
when expression: statement;
```

where the statement following the colon must be terminated by a break as in switch.

The expression following the `when` is a condition expression which may include an accept operation.

The ClassiC select follows a simplified execution pattern that is essentially sequential but maintains the effect of the simultaneous evaluation of the guards and non-determinate selection of one of them. A non-determinate choice is made as to which arm to evaluate first, and thereafter the arms are evaluated sequentially. The condition expression is evaluated. If it includes an operation which may involve synchronisation with some other process (an `accept` or a call of an entry for another process) a check is made to see if another process is waiting to rendezvous. If so the accept is taken and after execution of the entry by the waiting process the corresponding arm of the select is executed.

In the case that the condition expression, including the `accept` if present, evaluates false the next when statement is evaluated. If the end of the select instruction is encountered evaluation continues with the first arm. If no arm evaluates true, but there are one or more arms dependent on a rendezvous the process waits for the first such rendezvous to occur.

The ClassiC select does not contain a default arm as, for instance, in Ada or the C switch. If all guards evaluate false and there are no pending rendezvous then the program is terminated.

6.2.5 Process inheritance.

Since the ClassiC processes are simply active classes, they share all the inheritance properties of the inactive classes. This means that an active class may be derived from another active class, from an inactive class or, using multiple inheritance, from a combination of the two.

In the case where behaviour is inherited from another active class, clear rules are necessary concerning the order of activation. In fact, those rules are precisely the same as those defined for an inactive class. The constructors of the base class or classes will be called sequentially. Initialisation will be performed before the base constructor executes the `coreturn` statement allowing the next constructor to be called. The behaviour of the object will be provided by the processes for both the base and derived classes executing concurrently.

The second inheritance case is that in which an active class inherits from an inactive one. This is likely to be quite a common occurrence, with the inactive base class providing a data abstraction to which the derived class adds an activation process. If the base class has not been declared with a priority (and therefore has monitor semantics) there exists the possibility of concurrent access to members of the base class without the guarantee of mutual exclusion. It is therefore best for the programmer to ensure that all base classes have a declared priority by annotating the class heading appropriately.

The third case is that in which an inactive class inherits from an active one. This is also likely to be quite common, with the active class providing

some generic behaviour, for instance that of a buffering mechanism, while the derived class implements the buffer for some particular data type. In this case the monitor semantics of the base class are again needed to guarantee mutual exclusion between processes.

The issue of multiple inheritance of active classes is one which has caused several problems in the design of COOLs. The process instantiation method used in ClassiC provides a resolution of most of these problems, as is discussud in Section 13

6.2.6 Event handling.

The intention is to use ClassiC as an implementation language for embedded systems. As such it is necessary to provide some support for interfacing to external devices. Within C and C++ such concerns are left to the operating system, as is the issue of concurrency. In ClassiC concurrency is integrated into the language and device support needs to be as well.

As a 'low-level' high level language, C allows direct access to device registers (operating and memory management systems permitting) simply by pointer manipulation. Such a mechanism is about as satisfactory as any other way of low level device handling that has been proposed, and is retained for ClassiC. For event handling, however, something better needs to be done.

The mechanism used is the provision of a built in active class `Interrupt`. The constructor for class `Interrupt` takes as a parameter

the vector number (or other interrupt identification required by the hardware) and has an entry `virtual int operator ++()`. Instances of the class will accept this entry once only for each occurrence of the associated interrupt or event. It is of type `int` to allow the entry to be used in select condition expressions. It always returns the value *1*. The function is made virtual to allow it to be overloaded in the definition of derived classes. This can be used to allow broadcasting of interrupts, as is illustrated by the example below. To allow the derivation of multiple instances of classes derived from `Interrupt` without propagating interrupt handlers unnecessarily within the run time support `Interrupt` is also provided with a parameterless constructor. When this is called (for instance by instantiation of a derived class) no link to an interrupt is made.

Using this mechanism the statement

```
Interrupt clock(1);
```

creates an instance `clock` of the `Interrupt` class which is associated with interrupt vector 1.

```
Interrupt dead;
```

creates an instance `dead` which does nothing.

The statement

```
clock++;
```

suspends this process until the associated interrupt occurs.

6.2.7 Comparison of ClassiC with other concurrent and object oriented languages and environments.

C++

C++ is the base language for ClassiC, and is thus provides the core syntax for ClassiC. C++ is not, however, a concurrent language. Any support for concurrency must be explicitly programmed using the basic sequential operators of the language.

Concurrent C++

Concurrent C++ was produced by merging the concurrent extensions used for Concurrent C with C++. As such the concurrent parts of the language are orthogonal with the object oriented features, and the language cannot be classified as a COOL. The concurrent extensions are similar in design to the concurrent feature of Ada, and are therefore similar to those in ClassiC. However, since a process is not associated with a class, there is no process inheritance and no concept of active objects.

Ada and Ada 95

Ada as originally specified is a concurrent language, with a model of concurrency similar to that used in ClassiC. Ada95 added the ability to build derived data types in the form of the "tagged record". When combined with the existing facilities for encapsulation and data abstraction this has led Ada95 to be described as an object oriented language, although it might more correctly be described as a being a language which more easily

supports an object oriented style of programming than its predecessor. However, these facilities do not work together in an integrated manner, so it does not qualify as a COOL and the majority of design issues discussed in this paper do not apply.

Eiffel and Eiffel//

Eiffel is a 'pure' object oriented language designed by Meyer. Meyer's concurrent development of Eiffel includes concurrent extensions based on a processor, rather than a process, based paradigm. Eiffel// is another concurrent extension of Eiffel which includes a process model of concurrency, with objects inheriting activity from a special class PROCESS. The consequences of such a design choice as opposed to the *coreturn* method used in ClassiC, have been discussed above.

ACT++

ACT++ is an extension of C++, in the form of a class library, that provides concurrency following the actor model of concurrency [Agha, 1986]. In this model the active objects are *actors*, which each have a set of different *behaviours*. Active objects process messages concurrently, and after processing each message adopt a replacement behaviour. The process state of each active object is encapsulated in its current behaviour. The similarity of the actor model to that of active objects communicating by message passing has been noted [Bal, Grune, 1994]. The difference is that the actor model enforces a particular discipline on changes of process state, as outlined above, whereas the process model used in ClassiC and

others follows a more conventional programming model, with program state determined by object variables.

Parallel libraries and operating systems.

This group of languages, libraries and operating systems are considered together. They share their roots in the parallel programming, rather than the concurrent programming community. While parallelism and concurrency are often stated to be synonyms as, for instance by Burns and Welling [Burns, Welling, 1990], the choice of term is indicative of a clear difference of concerns. The parallel programming community is concerned with achieving maximum parallelism of computation. Within such languages synchronisation is a secondary concern, required to ensure the correct operation of parallel algorithms but essentially viewed as an obstacle to the primary aim of parallelism. Parallel languages, libraries and operating systems such as PVM [Benguelin et. al., 1990], Linda [Carriero, Gerlernter, 1989], CHAOS [Hwang et. al. 1995] and CHAOS++ [Chang et. al. 1995] concentrate on the easy spawning of parallel threads of computation. Synchronisation tends to be more cumbersome, and efficiency of execution will tend to dominate over considerations of language structure and consistency. Concurrent languages, such as ClassiC, have been designed for systems whose behaviour includes concurrency. Here the major concern is temporal behaviour, and therefore synchronisation is extremely important. The emphasis in the design of these languages is concise expression of communication and synchronisation patterns, sometimes at the cost of an efficient parallel implementation. In the case of ClassiC, the

encapsulation of code and data and the small scale nature of a process would be a considerable obstacle to the construction of very high performance parallel versions.

POOL

POOL is an object oriented language designed for parallel programming (and hence concurrency). It has the unusual characteristic that all objects are considered active, and that the notions of subtyping and inheritance are separated. Unlike ClassiC it has introduced a completely new syntax and is not a derivation of an existing, commonly used language.

Java

Java, although now promoted as an applications language for the Internet, was originally designed for use in embedded systems. The language supports concurrency, using the common COOL method of inheritance from a special active class. The relative merits of this method in comparison with that used for ClassiC have been discussed above. Like ClassiC, Java was derived from C++, with influences from other languages, but although much of the syntax is similar there are quite profound differences in the semantics of the language and many features have been deleted. ClassiC is a pure superset of C++.

Other languages

Concurrency and object-oriented programming are two active areas of research, and as such it is not surprising that many languages covering

Other languages

Concurrency and object-oriented programming are two active areas of research, and as such it is not surprising that many languages covering these areas have been proposed. There are also many concurrent operating systems and kernels, including many that are said to be "object based". This paper considers only programming languages. Those not discussed in detail above may be classified as follows.

Languages that support concurrency but not object orientation include Algol68, Mesa, Concurrent Pascal, Modula-2, occam, Ada, SR and many others. Object Oriented Languages that do not support concurrency as an integral language facility include Smalltalk, C++, Eiffel, Objective C and several others. Object Oriented Languages which do support concurrency, but not in a manner integrated with the class system, and therefore not qualifying as COOLs, include Modula-3, Oberon and Concurrent Smalltalk. Readers are referred to [Bal,Grune, 1994] for a more complete consideration of these many languages.

6.2.8 An illustration.

This section develops an example of ClassiC programming in order to demonstrate the features of the language and their use. The example selected is a clock process which provides delay, wakeup and "tick" functions, deriving its timing from a regular clock interrupt.

The first definition is for a class `tick` which provides a regular tick. This class is derived from the built in `Interrupt` class so that the clock can be propagated along a chain of similar objects.

```
#include <bool.h>
class tick(1): Interrupt{
public:
     tick(int count = 1);            //Count gives entry:
     virtual int operator ++();      // for next tick
     void operator @() {};           //Tick propagated
protected:
     static Interrupt* chain = 0;
}
```

This completes the definition of the derived class. The constructor takes as a parameter the period of the tick required, with a default of *1*, the same period as the hardware clock. The new class overloads the ++ operator, providing a new source of tick interrupts for future instances of tick. Client processes wait for ticks by executing the unary @ operator on the instance of the tick. The @ operator is defined within the declaration. Since it has nothing to do except synchronise, the function body is empty. The ++ operator must in addition return a value of *1* so as to be compatible with the Interrupt ++ operation. It is defined below.

```
int operator++() {
     return 1
}
```

The body of the constructor, which is also the main body of the process is shown below.

```
tick::tick(int count) {
     Interrupt& source;              //source of ticks
     if (chain == 0) {
          //    first instance of tick
          Interrupt clock(CLKVEC);
          source = clock;
     } else {
          //    previous instances
          source = *chain;
```

```
        }
        chain = this;           //link for next instance
        //    initialisation finished
        coreturn;
        //    remaining text executed by new process
        int t = 0;
        bool event = FALSE;
        do {
              // main loop of new process
              select {
                    when (event&&accept(operator++())):
                          //    clock tick propagated
                          event = FALSE;
                          break;
                    when (source++):
                          //    tick from source
                          event = TRUE;
                          t == MAXINT ? t = 0: t++;
                          break;
                    when ((t>count)&&accept(operator@()):
                          //    divided tick to client
                          t -= count;
                          break;
              }
        } while TRUE;
}
```

The initialisation is entirely to do with chaining through the hardware clock ticks for other instances of the `tick` (or derived) types. The main loop consists entirely of the select statement, one arm of which serves each of the external interfaces. There is no need to provide a termination condition for the loop. It will terminate automatically when the destructor (in this case the default destructor) is called.

The definition below shows how a time of day clock may be constructed using the `tick` class, but derived from an inactive class.

```
class Time {
public:
     Time(int h = 0,int m = 0, int s = 0);
     operator++();            // increment one second
     //... other access functions
}

class TimeOfDay (0): Time {
public:
```

```
        TimeOfDay(int h = 0,int m = 0, int s = 0);
}

TimeOfDay::TimeOfDay(int h, int m, int s):
        Time(h,m,s)
{
        hours = h; minutes = m; seconds = s;
        coreturn;
        //      child process starts here
        tick secs(TICKSPERSECOND);
        do {
                @secs;
                operator++();       //explicit call of own
                                    // operator function
        } while TRUE;
}
```

Here the inactive class `Time` has been endowed with activity by the derived class `TimeOfDay`. Instances of `TimeOfDay` have the same properties as those of `Time` because they share the same access functions but a `TimeOfDay` has the added property that it tells the time.

6.2.9 Scheduling issues.

The language definition of ClassiC makes no assumptions about the underlying process model, except that processes have some initial priority Since there is, as yet, no way defined to change that priority, the priority model is static. This may well change as the language develops, particularly if it proves to be attractive for the implementation of real time systems.

The language as defined is entirely conventional in its underlying process, inter-process communication and synchronisation models, and as such is likely to be as subject to already identified scheduling problems such as process starvation, priority inversion, deadlocks and so on as any other such language. By the same token, the established methods for

dealing with these issues are applicable also to programs written in ClassiC.

6.2.10 Derivation of active classes.

COOLs follow a conventional concurrent sequential model of computation. Each active object has associated with it its own independent thread of execution. Generally the execution of the thread begins on instantiation of the object, with a special method forming the program for that thread of execution.

One design problem that remains is what to do when a new class is derived from an existing active class. There are a number of considerations that need to be taken into account. We consider a number of possibilities.

1. An inactive class is derived from an inactive class. This is the normal object oriented derivation. Both classes are merely abstract data types and no change in state can occur except within a method call.
2. An active class is derived from an inactive class. Here a new thread of activity must be provided for the active derived class. Presumably this thread of execution will access class members defined by the base class.
3. An inactive class is derived from an active class. The resultant object is, surprisingly, active, the activity being provided by the process supporting the base class. In this case the derived class

may provide additional data members and additional methods, or overload existing methods.

4. An active class is derived from an active class. This is the most difficult case, because the activity of the new class presumably includes the activity of the base class and that of the derived class.

To provide for this derivation some way must be found to combine the two supporting processes. If this is not done the programmer will have to provide a completely new process body for the new, derived, class - in the process losing many of the advantages of derivation. This problem has been observed within the design of several concurrent object oriented languages and has been named the "inheritance anomaly" by America and others [Matsuoka et. al. 1993]. As will be seen below, the design of ClassiC provides an elegant solution to this problem.

The type inheritance model of a COOL is likely to allow multiple levels of inheritance and multiple inheritance, allowing a new class to be the leaf of an inheritance tree that could include a mixture of both active and inactive classes. Any solution to active class inheritance must address this situation as well.

The problems associated with case 1 are simply the well known ones of controlling shared data. Once these have been overcome then they provide a solution to case 2 as well, since all this adds is one new process. Thus the access pattern is the same as the generic one for multiple processes accessing the same object. In the design of ClassiC this is

covered by guaranteeing that method calls of any object which change its state are atomic, guaranteeing mutual exclusion between processed calling methods of a common object.

Case 3 provides no additional concurrent activity, so there are no additional problems in terms of synchronisation or mutual exclusion. The additions can make no changes to the activity of the class, otherwise this would be an active extension of the class, so any extensions are limited to additional methods or data members whose state changes only in a method call. The addition of methods requires that the supporting process be augmented in some way to cater for the extra methods.

Case 4 is the most difficult case, since it is not obvious how to provide a new supporting process for the composite object that mixes the activity of the derived and base object in a sensible way. The most usual solution is to overload the process body, requiring the programmer to produce a completely new body for the derived class, which includes the required activity from the base class. While this is a simple solution for the simple case, in the case of extended derivation, where the derived class is the leaf of a large derivation tree the programmer is left with the task of re-implementing the activity of all the active classes from which the new class is derived (the inheritance anomaly). This is not far from re-implementing the whole class, so the value of derivation is marginal.

6.2.11 ClassiC solutions.

The solutions adopted in the design of ClassiC come about as a natural extension of two of its basic characteristics. The first is the use of synchronous method calling discussed earlier. The second is the way in which active objects are instantiated. ClassiC is an extension of C++. The constructor for a class serves for both initialisation and as the body for an active class. After initialisation a coreturn instruction causes a program fork. The parent process executes a return from the constructor while the child continues execution of the rest of the constructor, which forms the body of the support process for the active object.

The semantics of C++ derived class instantiation dictate that the constructor(s) for base classes will be called in turn before the constructor of the derived class. The simple application of this to the constructor as modified for ClassiC provides a solution to the derivation of active classes from other active classes, in the process avoiding the inheritance anomaly.

Below are schematic outlines for two active classes.

```
class base {
public:
     base() {
            baseinit();
            coreturn;
            basebody();
     }
entry:
     ...
}

class derived: base{
public:
     derived() {
            derivedinit();
            coreturn;
            derivedbody();
```

```
        }
    entry:
        ...
    }
```

Consider an instantiation of the class `derived`. Firstly the constructor of the base class, `base()`, will be called. This will proceed to initialise the object by calling `baseinit()` and then execute the `coreturn`. The parent process will return and proceed to execute the constructor `derived()`. Meanwhile the child process will execute `basebody()`, providing the activity for the base class. Execution of the constructor of the derived class may assume initialisation of the base class. In addition, any methods called in the base class will be supported, since the base object is already active. Execution of the `coreturn` causes a second process fork and return of the parent process. The end result is that the derived object's activity is supported by two concurrent processes, one supporting the base part, the other supporting the derived part. These two processes obey the normal rules of inter-process communication, so the derived process may access the base class by synchronised method calls in the normal way. The programmer of the derived class has only to provide a process body defining the modified behaviour of that class and the inheritance anomaly does not arise.

6.2.12 Producing a buffered derivative of a class.

The type inheritance features of ClassiC can be used to provide asynchronous method calls in a number of different ways. The required non-synchronisation can be introduced into the call itself, by programming an intermediary class which buffers parameter values and executes a

method call allowing the original caller to proceed while it waits for the called method. Where a value is returned a similar "wrapper" class can be used to pass the result back to the caller while the called object continues. A combination of the two can also be used.

A "wrapper" class for an existing base class is designed as follows:

```
class base {
public:
     base();
entry:
     virtual m();
     virtual m1(int i);
}
```

The derived "wrapper" class has the same entries (concurrent method calls), and is defined as follows

```
class wrapper: base {
public:
     wrapper(){
            coreturn;
            for (;;) {
                   select {
                   when accept(m): base::m(); break;
                   when accept(m1): base::m1(ti); break;
                   }
            }
     }
entry:
     m(){}
     m1(int i){ti = i;}
private:
     int ti;
}
```

The wrapper class, derived from the base class, overloads its methods with methods which do nothing but call the corresponding base class method. They perform the call within the main loop in the constructor for the wrapper class. The calling process can proceed without waiting for the base class process to be accept the rendezvous. The wrapper class is used precisely like the base class, so a blocking method call

```
base b;
...
b.m()
..
```

is replaced by a non blocking call

```
wrapper w;
...
w.m()
...
```

The derived class has the same behaviour as the base class but its supporting process buffers method calls allowing the calling process to continue even though the process base may not be ready to handle the method call. This example is singly buffered, although it is possible to produce multiply buffered versions at the cost of complexity.

6.2.13 Analysis of the ClassiC rendezvous.

If an object has a supporting process associated with it then it is an active process. In this case calls to the methods of that object take on the nature of a rendezvous between processes. The calling object is delayed until the called object executes an accept instruction specifying the method called by the other process. The called object is then suspended until the calling object has completed execution of the method, or, in an alternative view the called object is interrupted and executes the method while the caller is suspended. Both views are equivalent, the former perhaps easier to visualise in terms of program flow, the latter being more useful from the point of view of program analysis, as will be seen below. Subsequently both objects continue execution concurrently.

The method is defined precisely as a normal C++ method, or class member function, with the body of the rendezvous being provided by the

member function. The only addition to normal C++ is the accept instruction. The call of the method happens within the scope of the called object, just as is the case in standard C++, so during that call the method object has access to the private and protected members of the called object. This provides an inter-process communication model that is very similar to the Ada rendezvous, with one important exception. In the ClassiC rendezvous the program text of the "entry" resides in the member function of the object whereas Ada has an extended syntax for the entry in which the program text for the entry resides within the entry block. As explained above, this difference produces an important simplification with respect to exception handling.

Just as the Ada rendezvous is supplemented by a select guarded alternative command so is the ClassiC rendezvous. The ClassiC select allows a number of program arms, each guarded by a condition which may include one or more `accept` instructions. If an accept is matched by a call of the corresponding method it evaluates to true, a value which is available to form a part of a boolean expression in the guard. Should more than one guard evaluate to true then an indeterminate choice is made as to which guard to execute. Should no guard evaluate to true but there is at least one guard containing an accept instruction then the object will be delayed until there is a matching method call. If there are no guards that evaluate to true and there are no guards containing an accept instruction then the object terminates. In the absence of accept instructions the select instruction is equivalent to the Dijkstra guarded *if* [Dijkstra, 1976]. Should one or more of

the guards contain an accept instruction then it is equivalent to the select instruction in Ada.

Analysis of this kind of rendezvous is well understood. The analysis given here is based on that by Andrews [Andrews, 91]. A rendezvous occurs when a calling or client object calls a method *m* in a server object supplying a parameter list *p* and that server object executes a matching select. Method *m* has a formal parameter list p_f, contains an instruction list *S* and returns the value *r* which is assigned to *s* in the client. This rendezvous can be simulated using synchronous message passing, where the method call

```
s = server.m(p)
```

is simulated by the sequence

```
server ! m(p); server ? s
```

where *!* is a synchronous message send operation to the object server and *?* is a matching synchronous receive operation as used in CSP.

Using Hoare programming logic [Hoare, 1969] we can write a triple for the axiom for the call

Rendezvous Call Axiom:

```
{ U } s = server.m(p) { W }
```

This axiom allows anything to appear in the postcondition W since the rendezvous may never complete thus in terms of a partial correctness proof for the sequential program client a postcondition false would be a valid result. Sound use of the axiom will depend on a satisfaction rule

encapsulating the interaction between the two objects. By writing in the message passing simulation a proof outline is obtained.

```
{ U }  server ! m(p) { V } server ? s { W }                    (1)
```

Considering now the server, its part of the rendezvous is the execution of the accept instruction followed by the instruction list *Sm*, contained in the method *m*. Again we can write a simulation using message passing.

```
client ? m(pf); Sm; client ! r
```

If P is the precondition of the accept instruction and Q is the postcondition then we can write a proof outline for the message based simulation of the accept instruction.

```
{ P } client ? m(pf) { R } Sm { T } client ! r { Q }      (2)
```

Once again we are unable to say anything about the assertions R,T and Q, since the message operations may never complete. We await the satisfaction rule for the rendezvous.

We should also consider the more general case in which the accept instruction appears as part of a guard in a select instruction. In this case we use the inference rule for the select instruction.

Select Rule:

$$\frac{\{R_i \wedge B_i\}\ S_i\ \{T_i\},\ 1 \le i \le n}{\{R\}}$$

```
      select {
      when accept(m1) && Bi: Sm1; S1; break;
            ...
      when accept(mn) && Bn: Smn; Sn; break;
      }
```

$\{ T \}$

For execution of one arm of the select we can use the select rule and proof outline (2) to obtain a further proof outline.

$\{ P \}$ client ? m(p_f) $\{ R \wedge B \}$ Sm $\{ Tm \}$ client ! r $\{ Q \}$ (3)

Note that *Tm* is the post-condition of the statement list *Sm* contained in the method definition, and that we also have

$\{Tm\}$ Sn $\{T\}$

The matching message operations in (3) and (1) must satisfy the satisfaction rule for synchronous message passing, which is that for all such matching pairs of communication instructions it must be shown that

$$(X \wedge Y) \Rightarrow (C \wedge D)^{x}_{y}$$

where *X* and *Y* are the preconditions of the receive and send instructions respectively and *C* and *D* are their postconditions, *x* is the target of the receive and *y* is the value supplied to the send. As well as this it must be shown that the assertion *V* in (1) is implied by *U* in (1) if *p* is assigned to p_f. If this condition is satisfied then application of the satisfaction rules for the two matching pairs of communication gives the satisfaction rule for the rendezvous as :

For every pair *server.m(p)* and *accept(m)* show that

$$(U \wedge P) \Rightarrow (R \wedge B)^{p}_{fp} \wedge \quad (V \wedge T) \Rightarrow (W \wedge Q)s_{r}$$

The other requirement is to show that the proofs of the two objects are interference free. Techniques for avoiding interference include the use

of disjoint variable sets for the two objects. In fact the encapsulation of classes in C++ (and therefore ClassiC) guarantees that the variable sets of two objects are disjoint so long as global variables are not used and the static attribute is not employed for class members. If either of these conditions is breached then it will be necessary to employ global invariants within the proofs for the two objects.

6.2.14 A Proof Example

The example above showed how a wrapper class may be used to loosen the synchronicity of the ClassiC rendezvous. This will be used as an example to show how the proof techniques discussed may be applied to the ClassiC rendezvous.

The objective is prove that a call of `wrapper::m` or `wrapper::m1` will always result in a call of `base::m` or `base::m1`. There are two rendezvous involved, that between the caller of the wrapper as it executes `w.m()` and that between the wrapper and its base class when it calls `base::m()`.

The call `w.m()` is simulated by message passing to give the proof outline, from (1),

```
{ U }  w ! m() { V }  w ? void { W }
```

Since `m()` is a void function the result list is not used (received into void). The wrapper object must execute the matching `accept` statement to complete the rendezvous. This gives the proof outline, from (3),

```
{ P } main ? m() { R } ; { T }  main ! void { Q }
```

since the guard condition B is true in this case. To show that the rendezvous occurs and that `synch::m()` is called we need to establish the satisfaction rule

For every pair `server.m(p)` and `accept(m)` show that

$$(U \wedge P) \Rightarrow (R \wedge B)^{P}{}_{fp} \wedge \quad (V \wedge T) \Rightarrow (W \wedge Q) s_r$$

Since the parameter list is empty, the returned result is not used, the condition B is true and $T = R$, this reduces to

$$(U \wedge P) \Rightarrow (R) \quad \wedge \quad (V \wedge R) \Rightarrow (W \wedge Q)$$

Now R is the precondition for the select instruction, which in this case is simply the precondition for the `for` instruction and is true so long as the object `w` has been initialised, which occurred with execution of the constructor, thus R is true and $(U \wedge P) \Rightarrow (R)$. In this case, where no return value is sent, the statements `w?void` and `main!void` can cause no change of program state so $(V) \Rightarrow (W \wedge Q)$. A similar analysis can be performed for the single parameter case, `m1`, although here account must be taken of the assignment to t_i.

6.2.15 Absence of deadlock

Although synchronous communication apparently makes deadlock more likely it does have the advantage that the incidence of possible

deadlock is localised to the parts of the program which communicate. In the case of ClassiC this is the method calls and `accept` instructions. Deadlock will not occur so long as every method call is matched (eventually) by an `accept` instruction. To show absence of deadlock it is necessary to show that executions of an `accept` instruction for a method occur the same number of times as the calls of that method. This may be relatively straightforward in the case where communication is limited to a pair of objects. It is likely to be more complex in the case of server type objects which serve a large number of clients or where the accept instruction is embedded in a select instruction. Many servers are likely to have both characteristics, with the body of the object consisting of a select statement enclosed by an endless loop, as shown below

```
for (;;) {
     select {
            when accept(m1): service1() break;
            when accept(m2): service2() break;
            ...
     }
}
```

In such a program it is necessary to show that each arm of the select will run to completion to guarantee that every call will be serviced. Once again, where the statement lists include calls of methods of other active objects the situation becomes more complex. Deadlocks can occur in cases where those objects are ultimately dependent on the server. While such situations are susceptible to analysis this can become very complex. The author is investigating the use of visualisation aids to help in this task. In any case, the task is made easier by the localisation of communication inherent in the rendezvous model.

6.3 Translating CCS specifications into ClassiC

It will be seen from the above description of ClassiC that it does contain most of the characteristics that are necessary for implementation of CCS based hypermedia systems. This section will show in more detail how the translation may be made.

6.3.1 Events

Display events.

A display event is mapped to a call of the the display method of an instance of a class that displays the appropriate type of object. There is no defined environment for ClassiC, so some appropriate display toolkit will need to be provided. Since ClassiC is derived from, and link compatible with, C and C++, toolkits available for those will be usable but are generally operating system dependent. The examples given here use an adaptation of the Abstract Windowing Toolkit (AWT) that is part of the Java language environment [Sun, 1996] but is modified to work with a concurrent, rather than event-based interaction style. In the event based style, as operated by Java, the X window system toolkit [Nye, O'Reilly, 1990] and several other windowing systems, program control is invested in a hidden "event loop" within the user interface system, with the program being structured as a set of event handlers. In the concurrent style the user interface is operated by an explicit process which communicates with user processes using normal

inter process communication mechanisms. In ClassiC the *graphics context*[1] could be implemented as an active object, as follows.

```
Class GC {
 Public:
 GC(pixmap b ...);
 ~GC;
 boolean drawImage(iclass Image, int px,py,sx,sy,
                                      GC & where);
 Boolean postControl(controlType c, void (*response) ())
 ...
}
```

Ellipsis (...) has been used to indicate detail that has been omitted as not pertinent to this discussion. In the GC defined above the constructor, GC, is provided with a pixel map buffer and other initialisation information. This GC is an *active object*, so the constructor will produce a process fork by executing a `coreturn` instruction, leaving a process running to handle the graphics context. Other processes communicate with this using the methods such as `drawImage` and `postControl`. Their function is explained more fully in the sections below.

Using the a graphics context `g` as defined above, the translation of the display event becomes

```
Image = getImage(<imageURL>);
boolean b = g.drawImage(Image,px,py,px+x,py+y,this);
```

[1] "Graphics context" is a term borrowed from the X window systems and encapsulates the state of a particular display area, in X it operates in the X server and by caching the state locally avoids unnecessary network traffic. The grouping of all relevant display state into one object has proved to be useful enough that the use of graphic contexts is now common in graphics packages whether or not they are network based in the same manner as X.

Here the first line initialises an Image object containing the image data referred to by the Uniform Resource Locator (URL). The second line renders the image at the location in the display window given by `px, py` with a size `x,y` pixels, using a method, `drawImage`, of the current graphics context `g`. Both `getImage` and `drawImage` are polymorphic, in that they will handle a wide range of displayable data, including still and moving images. Sound data will not be handled by this combination of methods, which means that sound based events will require to be separated out and realised using their own access and "display" objects.

User events and input events

User events come in the form of posting interaction controls that will at a later time be responded to and will cause input events. A class is defined to post control buttons

```
control = new Button(label);
boolean b = postControl(control, &response);
```

The first line initialises the control, in this case a button, the second line posts the control to the GC process, causing it to display it. The second parameter is a pointer to one of the current process' entries, which will be used to notify a control action by the user.

The input event is handled simply by waiting for the call of that entry, by executing an `accept` statement, as follows.

```
accept(response);
```

This will cause the process to wait until the response entry has been called by the GC process after which it will continue at the next statement.

Synchronisation events

Synchronisation uses a similar mechanism, simply utilising the synchronised rendezvous that is provided by ClassiC. This if two agents, `agentA` and `agentB` need to synchronise on an event pair $e, \bar{e}$, then an entry, `e`, in *agentA* is used to represent the pair. Agent B represents $\bar{e}$ using a call of the entry, as follows.

```
AgentA.e();
```

AgentA represents the other half of the event *e* using an accept statement.

```
accept(e);
```

Since the rendezvous is synchronised, synchronisation between the two agents will occur.

6.3.2 Prefix, the '.' Operator.

The '.' Operator represents sequential composition. In a sequential programming language such as ClassiC this is represented simply by using the instruction sequence of the language.

6.3.3 Agents and agent definitions.

A CCS agent is composed of a composition, both parallel and sequential, of sequential agents. If these agents are to be composed in parallel they will require independent processes to run them, which in ClassiC requires the use of active objects. A single agent definition may be instantiated both in sequential and parallel composition. Fortunately ClassiC allows both active and passive variants of the same class, and so this can be achieved. The following shows an outline of a class to represent an agent.

```
class agent {
 public:
 agent(boolean active);
 {
     if (active) coreturn;
     agent_body();
 }
 private:
 agent_body();
}
```

The sequence of events that makes up this agent is coded in `agent_body`. In order to create an instance of the agent a variable of type `agent` is created, and instantiated with using the appropriate constructor parameter to signify whether it is active or passive. Sequential composition of agents will use passive agents. Thus the CCS

agentA.agentB

would be represented using two class definitions, styled on that above, and instantiated as follows.

```
AgentA aa=false;
AgentB ab=false;
```

Here the assignment to false causes the constructor to be called with a parameter value of false, which reduces to a call of the `agent_body`, so the above is equivalent to

```
AgentA::agent_body; AgentB::agent_body
```

which is simply sequential composition of the two agents.

6.3.4 The parallel composition operator '|'.

Having defined the dual purpose agent class above, we can us it to produce parallel composition. The CCS *AgentA|AgentB* is represented by the following.

```
AgentA aa=true;
AgentB ab=false;
```

The AgentA constructor is called with a parameter value of *true,* and thus is equivalent to

```
coreturn;
agent_body();
```

The execution of `coreturn` will cause a process fork. While one process executes `agentA`'s `agent_body` the other, on return, executes `AgentB`'s constructor with a parameter value of false, causing a direct execution of `agentB`'s `agent_body`. Thus we have the two agents acting concurrently.

6.3.5 Choice, the '+' operator.

The CCS choice operator is modelled on the Dijkstra multi-armed "if" statement [Dijkstra, 1976], as is the ClassiC `select` instruction. The CCS '+' operator results in an agent being guarded by its first event, while the ClassiC select separates out the guard event from the body of the select arm. To use the ClassiC select the CCS agents will require to be reconstructed in a similar fashion. The following CCS expression

agentA+agentB

will need to be recast as

headA.tailA+headB.tailB

where *head* represents the first event in the agent and *tail* represents subsequent events. These events will be input events, which are represented by entry acceptances, so the above translates to the following ClassiC.

```
select {
when accept headA:
      tailA ta = false;
      break;
when accept headB:
      tailB tb=false;
      break;
}
```

The first entry to be called, *headA* or *headB* will cause the corresponding arm of the select statement to be executed, resulting in the execution of the tail of the agent.

6.3.6 Association, the '(' and ')' operators.

The '(' and ')' operators are used to associate agents without naming them. In the examples given above named agents have been used as the operands of the various operators. Since Classic does not include lambda expressions, which would allow the construction of unnamed aggregates of instructions, the alternative of using named objects, where the names are meaningless, or simply automatically generated, will need to be used. So the agent *(agentA.agentB.agentC)* would be translated to the following.

```
class agent000 {
 public:
 agent000(boolean active);
 {
     if (active) coreturn;
     agent_body();
 }
 private:
 agent_body();
 {
    AgentA aa=false;
    AgentB ab=false;
    AgentC ac=false;
 }
}
```

Here *agent000* is an arbitrary name given to the agent. Otherwise is simply an agent containing the sequential composition of three agents, as described above.

6.4 Conclusion

This chapter has discussed how programming languages may be used to realise CCS based specifications, and has focussed in particular on one class of programming language, the concurrent object oriented

language (COOL). A detailed account has been given of one such language, ClassiC developed by the author. This language has particular characteristics that make it a suitable vehicle for realising CCS specifications. Examples of translation from CCS operators and agents to ClassiC code have been given, and the process can be seen to be generally straightforward. This simplicity of translation is due to several characteristics of the language. Firstly, the language provides a process based model of concurrency and fits well with a process based model of user interface construction. Both of these are a natural fit with a process algebra such as CCS. Secondly, ClassiC contains the necessary construct to translate directly the CCS operators, namely a process fork (in the form of *coreturn*) and a choice operator (in the form of the *select* statement). Finally, and unusually amongst COOLs, ClassiC provides the means producing both active and inactive instances of a class, thus allowing classes to be used to represent agents without needing to duplicate definitions to allow for parallel and sequential composition. This is, in fact, an example of the absence of the inheritance anomaly. If the language suffered from this anomaly redefinition of the class would have been necessary, simply because of the possession of activity by one of the variants of the class.

Chapter 7: Java, JavaScript and HyTime

The previous chapter has shown that CCS hypermedia specifications may be translated fairly straightforwardly into the concurrent object-oriented language ClassiC. Although as a language it has a suitable structure and semantics to be a good vehicle for implementation of these systems, ClassiC does suffer from a major shortcoming, the lack of acceptance of the language as a standard programming or scripting language. The importance of openness in hypermedia has been put forward in many influential works, starting with from [Halasz, 1988]. Reliance on a non-standard, or not widely accepted, implementation method would severely prejudice the chances of acceptance of any design method. However, the method is not dependent on any particular implementation language, and there are languages widely accepted as implementation vehicles for hypermedia systems that may be suitable for realising CCS specifications, although, as we shall see, the code produced may be less elegant than that produced using ClassiC. Three possible languages are discussed here, Java, JavaScript and HyTime.

7.1 HyTime

HyTime is considered in this chapter because it is a format for hypermedia description that includes description of temporal behaviour, and is in that sense closer to a programming or scripting language than other markup formats which seek solely to describe content and connectivity. HyTime is a standardised hypermedia structuring language for representing hypertext linking, temporal and spatial event scheduling, and synchronisation. The HyTime approach to ordering and synchronisation is essentially declarative, as opposed to the imperative nature of process algebras and programming languages. HyTime separates structure from content information, and included within the structure information can be attributes that declare an object's existence in a temporal, as well as spatial frame. The temporal frame can be according to a reference frame or with respect to other objects, allowing sequence and duration to be expressed. As such, HyTime forms a more difficult target from process algebra based specifications that traditional imperative scripting and programming languages.

7.2 Java

The programming language Java [Sun,1995] has recently received much attention as a standard implementation language for World Wide Web applets. The ambitions of Sun Microsystems, the language's originator, spread much wider than that, however. The previous chapter has argued that they are suitable implementation vehicles for a wider range

of “serious” multimedia documentation, particularly technical documentation.

The origins of Java lie with C++ and C. Java includes facilities to handle concurrency, and is a purer object oriented language than its ancestor, C++ (or, for that matter, ClassiC).

When considered as a language that is amenable to formal analysis Java is a huge improvement over both C and C++. The Java language has jettisoned almost all of the language features that made C so "dangerous"[Sun, 1995]. Experience has shown that it is just these features that make programs difficult, if not impossible, to verify formally.

Some features of Java would seem to be unsuited to formal analysis. in particular all functions (or rather, object methods) may operate on object variables using side effects [Sun, 1996], and all parameters that are objects are passed by reference, and are therefore also amenable to modification. Of course, such features are part and parcel of the object oriented programming style, and Java is a pure object oriented language, in that facilities such as type definition and procedures and are only available within an object oriented context, that is in the guise of classes and methods thereof.

Meyer has proposed techniques of formal verification when using object oriented languages. The idea is summed up in the heading “design by contract”[Meyer, 1993]. Here the required state for correct operation of an object is encapsulated into an object invariant that is true after construction of the object by the constructor and must remain true after

each method call. In addition each method is defined by a required precondition and a postcondition. The obligations of the creator of the object are to ensure that each method maintains the object invariant and that it produces the required postcondition from the precondition. Likewise the obligation of the user of the method is to ensure the precondition. Under the normal laws of programming logic if all obligations are met the method will produce the correct results.

This method is also robust when used with the type inheritance facilities of an object oriented language. The invariant for a derived object is simply the conjunction of its own invariant and those of its ancestors. The principle is simple enough, although the complete invariant for a leaf class of a large inheritance tree might be a large and unwieldy predicate.

Meyer proposed this method for use with his own language, Eiffel, which includes mechanisms for assertions to verify the invariants and pre and postconditions built in. Java lacks these facilities (although assertions can be easily programmed if required) but in other ways is as pure an object oriented language as Eiffel, indeed many of its design features were derived from that language. Many of the features of Java make such methods simpler For instance, the restriction on multiple inheritance simplifies the construction of object invariants. The definition of standard behaviours by means of "interfaces" can similarly be included in the method. Since interfaces contain only constants and method signatures they introduce no new state and therefore require no additional invariants of their own. Classes that implement the interface must, however, ensure that

all implementations of the interface methods preserve the class invariant, which may capture the constant names declared in the interface.

The formal verifiability of Java programs is aided by the formality of the language definition. While this does not extend to a formal semantic model care has been taken in its design to ensure that operations, primitive data types and relationships within the specification are precisely defined

7.2.1 Concurrency

Java has taken the approach of creating active classes using inheritance of a special active class. Activity is bestowed either by inheriting the class Thread or by implementing the interface `Runnable`. The problem of multiple inheritance of active classes does not occur, since multiple inheritance itself is not included in the language. This does not, however, protect against the inheritance anomaly.

It is also necessary to provide a means of inter process communication. Java provides a low level mechanism, essentially using monitors. As has been noted in a previous chapter the coupling of monitor semantics with method calls can provide an inter-process communication method with very similar semantics to the Ada rendezvous. The requirement for this to be achieved is that the body of the active method serving the rendezvous be notified of the use of one of its methods by another object. While Java does not do this automatically, the low level facilities (essentially wait/signal) provided can be used to provide the same effect, but programmer discipline is obviously required to program the

correct operations. Programming in this style is discussed later in the paper. Also absent from Java is a non-deterministic choice instruction (as the Ada or ClassiC "select" or occam "ALT"), which greatly simplifies programming using such rendezvous. This can also be provided by the programmer, again at a cost in program complexity and syntactic inelegance.

7.2.2 Temporal properties

Java includes no explicit facilities for temporal assertion or programming of temporal properties. It does, however, provide delay and time-out operations that have a resolution down to one nanosecond! Using such real time modelling techniques as timed CSP specifications can be produced which could be refined to Java implementations. The difficulty would be to try and predict accurately the time performance of the run time system, particularly the memory allocation and garbage collection system. It would be possible to construct and realise a model that would be subject to run time time-outs and temporal errors due to additional overheads produced by the operation of the memory system. Fortunately, technical documentation systems are unlikely to have hard temporal constraints. As interactive systems they need to respond sufficiently quickly to not cause interaction errors, but the major temporal constraints that have been put forward earlier in this work have been to do with sequence rather than absolute time.

7.2.3 Java for large systems

The objectives of this work are to propose design and implementation methods for large multimedia technical documentation systems. The discussion above has covered in outline the suitability of Java so far as reliability considerations are concerned. The second issue is, is it a suitable implementation vehicle for large systems?

The keys to meeting these requirements are program structure and data abstraction. Object-oriented programming, the paradigm on which Java is based, is a development of data abstraction. It provides a means of packaging the data type and its implementation routines in a unified package and a mechanism, called inheritance, whereby new data types can be created by modification and extension of existing ones, rather than complete revision. This brings further advantages, in that data types have a greater consistency between functionally similar types, making the task of defining and using the consistent interfaces on which co-operative programming depends simpler.

In fact, Java is designed from the start around a model of co-operative programming in which the monolithic application programs of the past are replaced by an assembly of "applets", which can be loaded into a running program to enhance its functionality. The program level implementation of an applet is the basic Java object construct, a class. To allow applets to be loaded into running programs, Java programs operate as an assembly of different class definitions which are loaded from diverse sources as required. To allow the sources to be truly diverse classes are

given identifiers that operate within a global name space! This is a potential breach in the program security of Java, in that however well your own program has been designed and verified, if you import applets from external sources there are no guarantees as to their quality. The Java run time system takes measures to prove them against outright crashes or corruption of the software's operating environment, but does not guarantee correctness against any specifications – indeed, commonly there will be no specifications available. Addressing this issue is outside the scope of this work, and for the type of development that is envisaged here it is likely to be the case that the whole development will be firmly controlled by the company whose product the documentation supports. Sufficient to observe that the support for co-operative development offered by Java should be at least sufficient for the type of development situation likely for large documentation systems.

7.3 JavaScript

JavaScript is a hypermedia scripting language developed to be similar to Java by Netscape Inc. as a way of introducing interactive behaviour to World Wide Web documents that was quicker and easier to use than Java. JavaScript is defined as a scripting language rather than a programming language, which Java is. To the authors mind this distinction is spurious, so-called scripting languages are merely programming languages, albeit simplified and often unstructured ones. Being based on Java, JavaScript has the rare distinction of being a structured scripting language. For this reason it should be considered alongside Java as a

candidate for implementation of large multimedia databases. JavaScript is, in terms of the language rather than the implementation, a simplification of Java. As well as simplifying the language parts of the data model have been relaxed in a way that reduces the safety and verifiability of the language. The data model has been modified by relaxing the type checking and introducing a mixed data model in which some basic types of data are directly supported as first order values. The type checking is also relaxed and automatic coercions between types introduced. Object types are not defined as a class, but constructed dynamically by assigning values to the features of the object. Some basic object types (classes) are built in. Functions, separate from object methods, have been reintroduced. As a result of these modifications JavaScript is in many ways closer to C++ than Java, although obviously without the complexity of C++. As a scripting language, for small programs, it does not include support for co-operative development in itself, however it is designed to be embedded in HTML documents, which would allow the document to be used as the basic unit of modularity. HTML mechanisms such as frames would also be necessary to support concurrency, which is not directly supported in the language.

While such workarounds are possible, this type of compromised solution is hardly desirable as an implementation vehicle for systems for which a high degree of confidence is required. The conclusion must be that Java is a more suitable vehicle for this purpose than JavaScript.

7.4 Comparing Java, JavaScript and ClassiC.

The table below, adapted from [Bell, Parr, 1998] summarises the comparison between the three languages.

	Java	JavaScript	ClassiC
Object Orientation	OO only	None	OO and procedural
Inheritance	Single	None	Multiple
Templates	No	No	Yes
Concurrency	Active objects	None	Active objects
Thread instantiation	Inherit from Thread	None	Coretum
Inheritance anomaly	Yes, but no multiple inheritance, so not serious.	No concurrency	No
IPC	Shared variables (method calls)	None	Rendezvous (method calls)
Synchronisat ion	"Synchronised " (monitor) methods	None	Monitor classes
Non determinism	Event driven programming	None	Select statement
Memory model	Dynamic, garbage collection	Dynamic, interpreted	Static
Pointers	Call by reference, no explicit pointer arithmetic	No pointers	Call by reference, pointers, pointer arithmetic.
Modularity	Packages	None	Separate files, Unix conventions, namespaces

HyTime has not been included in this comparison. As a declarative language it is too dissimilar from the other three to be easily compared in a simple table. As has been observed earlier, an imperative language is a much easier target for refinement from a process algebra specification, so the choice falls between Java and JavaScript. JavaScript lacks many of the basic attributes necessary, such as modularity and concurrency, so the Java has been used in the following section illustrating refinement to this language from CCS.

7.5 Translation Rules to Java

7.5.1 Base Classes

The base class for all sequential agents is illustrated below.

```
class SeqAgent<x> extends Thread {
  private Graphics g;
  private Image image;
  private int x=0, y=0, px = 0, py = 0;
  public semaphore sem;

  public SeqAgent<x>(graphics gr, int ix, int iy, int ipx, int ipy) {
    g = gr;
    x = ix; y = iy; px = ipx; py = ipy;
    sem = new semaphore(0);
  }

  public void run() {
    <event sequence x>
  }
}
```

Figure 7.1: The sequential agent class

Each separate definition of a sequence (but not each instance) will require a new version of `SeqAgent<x>`, with the `<x>` replaced by a unique name and the appropriate event sequence defined.

The class variables are explained as follows: `g` provides the graphics context for display operations; `image` provides a handle for any image data (further variables will be needed for other types of data); `x` and `y` give the size of the screen area that this Agent controls; `sem` is a semaphore object used for communication with controls and synchronisation events and `px` and `py` give the position on the screen. The semaphore object is a slightly modified form of a standard semaphore in that it allows a non-zero value to be passed between the threads operating the wait and signal operations (here called `P` and `V` to avoid a name clash with the Java `wait`.

7.5.2 Events

Events can be subdivided into several categories. These are: *display events*, which cause the display (or replay) of some kind of data; *user events*, which create some kind of user control; *input events* which respond to user actions and *synchronisation events* which cause synchronisation between agents.

Display events.

A display event is mapped to a call of the appropriate display method of an instance of a specialised class that displays the appropriate type of object, using the appropriate data. Such classes form part of the normal

Java environment, so the display event maps simply to a call of this, preceded by an operation to load the image file.

```
Image = getImage(<imageURL>);
boolean b = g.drawImage(image,px,py,px+x,py+y,this);
```

Figure 7.2: Code for a display event

User events and Input events

User events come in the form of posting interaction controls that will at a later time be responded to. Because the two are so closely associated they are dealt with together here. The posting of a control is most easily done using an Applet object. Such an object type must be defined as in Figure 7.3.

```
Class UserEvent<x> extends Applet {
  private Button control;
  public void init(String label) {
    control = new Button(label);
  }
  public boolean action(Event event, Object arg) {
    if (event.target == control)
      seqAgent<x>.sem.P(1)
    repaint;
    return true;
  }
}
```

Figure 7.3: User event applet

The other component that is required is the code fragment in the Thread of execution to invoke this applet and waits for the input event. This is shown below.

```
UserEvent<x> ue = new UserEvent<x>("label");
sem.V();
ue = null;
```

Figure 7.4: Inline code to invoke a user event

In this example the code instantiates the user event, waits for a response and then removes the user event again, letting the garbage collection system clean up after it.

Synchronisation events

Synchronisation between two agents can only occur using a shared variable, which must be positioned in scope for both agents, possibly declared as a variable in the root class. The variable requires strong synchronisation. A suitable class definition to achieve this is the following strengthening of the semaphore class.

```
Class StrongSem {
  private flag = 0;
  public synchronised void P() {
    while (flag==0)
      try {wait();}
      catch (InterruptedException e) {}
    flag = 0;
    notify();
  }
  public synchronised void V() {
    flag = 1;
    notify();
    while (flag==1)
      try {wait();}
      catch (InterruptException e) {}
  }
}
```

Figure 7.5: The strong semaphore class

The class is used as follows. Consider two agents, one entering into *event*, the other into $\overline{event}$. The first agent uses a strong semaphore to represent the event, using its `V` method. The second uses the `P` method.

7.5.3 Prefix, the '.' Operator.

The '.' operator represents sequential composition. In a sequential programming language such as Java this is represented simply by using the instruction sequence of the language.

7.5.4 Agents.

Agents need to be classified as either belonging to the set of agents which must be capable of sustaining an independent thread of activity or those which are simply convenient groupings of actions. The former must be implemented within the body of a `SeqAgent<x>` instance, the latter may be defined as one of its methods and invoked accordingly. Where such an agent will require to be used in several different `SeqAgent<x>` class definitions it should be defined as a separate class and included as a variable in each.

7.5.5 Agent definitions.

Since agents are represented as method and class definitions the association of the agent name with its definition is automatic. Where name hiding is required *private* attributes on declarations can be used.

7.5.6 Choice, the '+' operator.

The provision of the choice operator is more complex. As discussed above, the use of the event driven paradigm in Java dictates that this role be taken by a hidden event loop. An extension of the technique used for the user and input events can be used to model the multi way choice instruction.

Figure 7.6 shows how this may be done.

```
Class MultiEvent<x> extends Applet {
  private Button control1, control2, control3...;
  private int buttonUsed=0;
  public void init(String label1, String label2, String label3...) {
    control1 = new Button(label1);
    control2 = new Button(label2);
    control3 = new Button(label3);
  }
  public boolean action(Event event, Object arg) {
    switch (event.taget) {
      case control1: buttonUsed = 1; break;
      case control2: buttonUsed = 2; break;
      case control3: buttonUsed = 3; break;
      ...
      default: repaint; return true;
    }
    seqAgent<x>.sem.P();
    repaint;
    return true;
  }
  public int branchTaken() {
    return buttonUsed
  }
}
```

Figure 7.6 : Multiple input event handling.

The user event handler now stores a value identifying the event which happened. The input event code in the main thread now simply has to interrogate this and execute a switch statement to make the choice.

```
MultiEvent<x> me = new
MultiEvent<x>("label1","label2","label3"...);
sem.V();
int tmp = ue.branchTaken();
ue = null;
switch (tmp) {
  case 1: ...
  case 2: ...
  case 3: ...
...
}
```

Figure 7.7: In line code to handle the choice

7.5.7 The parallel composition operator '|'.

Finally we consider the parallel composition operator. We can consider the production of a parallel composition operator to be a process fork. A system *A.(B|C)* only requires two threads of execution, since *A* has terminated before *B* and *C* are instantiated. Thus this can be modelled in Java by the creation of one new Thread, as follows

The abbreviated outline of the class for the new thread is shown in the top part of the figure, while the lower part shows the code in the original threat that invokes it.

```
class seqAgentC extends Thread {
  <header information as seqAgent<x>
  public void run() {
    <instructions for C>
  }
}
```

Figure 7.8: Implementing parallel composition

Note the requirement at the end for the `join` method call, to ensure that both threads have completed before another agent commences.

7.6 Including text, sound, animation and models.

The illustrations above have been restricted to the showing of methods for events displaying images and using buttons for interaction. Using the *AWT* (Abstract Windowing Toolkit) of Java the facilities exist to

use the same methods to invoke events handling text, sound and animation as well. The interaction library can include the most commonly used user interface techniques, including mouse position events which can be used to implement "hot links" embedded in text. The techniques for all of these are similar to those shown above, although, of course the detail of the implementation will be different. The Java libraries already include code to read the most commonly used data formats found in hypermedia systems.

The display and manipulation of 3-D and other models obviously requires a more sophisticated library of access libraries, but these are steadily becoming available in the guise of Java beans and CORBA IDLs. Using these the translation principles outlined above will hold good for these media as well.

7.7 Conclusion

The previous section has outlined how each major component of the CSP specification may be translated into Java code. The code produced is in places clumsy and inelegant, particularly when compared directly with the much simpler translations produced using the ClassiC language. This is due to the lack in Java of the three attributes which rendered Classic so suitable a translation target, the process based model of concurrency, and a choice operator and the means producing both active and inactive instances of a class. Additional complication is caused by two other properties of Java. One is the reliance on event driven, the second is the lack of a high level inter-process communication and synchronisation

mechanism, which results in the use of rather convoluted code to achieve these objectives.

Although some of the solutions are inelegant, and the code and definitions long winded in places this may not be a problem in practice, since the translation definitions are mechanical enough to be able to be automated. Working from these translation outlines it should be possible to produce an automatic code generator which will produce at least the framework of the system, leaving only the correct access methods for the media in use and the final decisions about screen layout and user interface techniques to be left to the user. If the storyboard notation were extended to include absolute definitions of data sources, screen layout and control use, then this could be automated as well.

Chapter 8: Conclusion

8.1 Results

The storyboard method of hypermedia design provides a design route that is both accessible to information design practitioners, being based on their current practice, and also provides a means by which documentation designs can be verified for adherence to a defined set of safety conditions.

It is clear from the investigations of the OMIMO project that hypermedia systems usage is beginning to penetrate into areas where they are indeed safety critical. Where embedded systems software is similarly safety critical there are already requirements for its design to be based on sound and verifiable software engineering principles, and it can only be expected that the same will be true of technical documentation systems. Much of the current work in the field of "industrial strength" hypermedia does not address this problem at all. The story board method, and the associated translation techniques to COOLs such as ClassiC and Java provide the basis for a design methodology which can produce soundly software engineered technical documentation systems.

The design of such documentation systems should be seen as a conscious process, taking into account the properties of the complete production as well as the component pieces of content. This is in contrast to the building of many hypermedia systems, which can be viewed more as a process of compilation. While appropriate for libraries and other collections, such a design approach is not suitable for self contained and highly directed and specified systems such as technical documentation systems.

8.2 Issues

Several issues have been raised by this work.

Much of the current research work on large, industrial strength hypermedia separated hyperbase structure from the dynamic content of individual objects within the hyperbase. It is argued here that in the case of safety-critical documentation systems complete control over temporal ordering is necessary, and therefore such a separation is undesirable.

The implementation method used produces a hypermedia system structured as a single program. With a large body of work directed towards system openness using layered, Dexter based system models, it is likely to emerge as a requirement for at least some technical documentation systems that they be structured in this way. It remains to be seen whether a task based design method, such as the one proposed, could be mapped to such a model. The question is posed of whether such layered models, although they do encourage abstraction of structure from content, can ever

produce the control of temporal properties that is required in safety critical applications.

As well as the program translation route put forward, there are other possible realisation methods. One would be to construct a CCS engine, along similar lines to the original Hypertext Abstract Machine, to execute the specifications directly. It should also be possible to make a translation to declarative formats such as HyTime, although it would require considerable work to be done to establish the equivalent semantics between the two systems.

8.3 Future work

As discussed above, there is considerable development work that still needs to be done on the storyboard specification method. For a start, construction of a demonstrator system using this technique would be valuable and would help to establish the potential of the technique, as well as the feasibility of the refinement process. Secondly, the technique itself requires further development. At its current state of development it is a very simple precursor to more powerful techniques, similar to the original versions of CSP or CCS. While the temporal and structural specification is probably sufficient, several aspects could do with enhancement. In particular, a more formal definition of content and the placement and location of controls and links would allow the refinement process to become more automatic. Also, in its current form, the method does not make any

explicit allowance for modularity. Such an enhancement would facilitate its use in larger development teams.

The storyboard specification technique appears to be a powerful way of bridging the "semantic gap" between the informal specifications of industry and the mathematical formalisms of computer science. It does this by combining a graphical, intuitive appearance with a formal content. It would seem to have many applications outside the field of hypermedia design. One of the author's colleagues is currently using storyboards to document test procedures for CAD software (this use of storyboards was devised by him independently of the work in this thesis. The storyboard notation outlined here could be used to provide a formal description of those test conditions, to be feed back into the design process. The accessibility of the storyboard idea makes it a good candidate for the specification of many dynamic systems, from human-computer interfaces to embedded systems based products.

Methods for realising such specifications need to be developed in more detail. The Java based method discussed here provides one route, as does the "CCS engine" discussed above. Translations into established formats such as HyTime, using data modelling techniques working from the formal semantics of CSP are also a possibility.

The method could also be developed into a fully-fledged "methodology". To do this would require the development of the supporting toolset. Some existing CCS analysis tools, such as the concurrency workbench (CWB) [Moller, 1992] could be used as part of his toolset, but

others require to be developed. In particular, a storyboard editor, syntax guided by the graphical syntax of the storyboard system and a content assembly editor, structurally guided by the CCS storyboard, could form a part of the toolset.

Within the current method the framing of safety conditions is still very hard work, since the workings of Hennesy-Milner Logic are quite obscure. If a similar storyboard technique could be devised to help with this the method would be much easier to use.

The storyboard specification and refinement method offers a great deal of potential for the specification of high reliability hypermedia documentation systems, but there is a great deal of work to be done to develop and establish it.

Acknowledgements

I would like to acknowledge the help given to me by a number of people who have aided and abetted me in this work and the final submission of this thesis. Firstly I must thank Dr. Jim Tabor, my director of studies, who has provided much sensible advice and has been ever conscious of the constraints involved in the pursuit of a part-time programme of study. I must also thank Prof. Glyn James for encouraging my registration and providing the budgetary support for this programme of study and also Prof. Clive Richards for his continued interest and encouragement in the progress of this work. I would also like to pick out some of my colleagues who have been particularly willing to act as a sounding board for the ideas that have gone into this work, and have also suggested avenues of study. They are Sam Porter, who has diligently read my drafts and given much useful advice, Frank Giannasi, who has also acted as one of my supervisors, Mike Poppleton and the late Gustavo Jara-Martinez. Finally, I should thank my wife, Edwina, and my family who have supported this work consistently, even when it meant that I wasn't doing other things that I should have been doing.

Bibliography

Ackermann, P., 1994. Direct Manipulation of Temporal Structures in a Multimedia Application Framework. *Proc. Multimedia '94*, ACM.

Ada 9X Project Office, 1992. *Ada 9X Mapping Document, vol.1: Mapping Rational, Intermetrics..*

Ada 9X Project office, 1992. *Ada 9X Mapping Document, vol 2: Mapping Specification. Intermetrics.*

Agha, C.H. 1986. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., USA.

Albertini, B., Anklesaria, F., Lindner, P, McCahill, M., Torrey, D. *The Internet Gopher Protocol: A Distributed Search and Retrieval Protocol.* Available at ftp://boombox.micro.umn.edu/pub/gopher/gopher_protocol/protocol.txt

Alexander, H., 1990. Structuring dialogues using CSP, in M.D.Harrison, H.W.Thimbleby (eds) *Formal methods in Human Computer Interaction*, Cambridge.

Alty, J., 1993. Multimedia: We Have the Technology But Do We Have the Methodology?, *Proceedings of EUROMEDIA '93* , Maurer, H. (ed), AACE , Orlando, Florida, pp. 3-10.

Alty, J.,Bergan, M. 1993. Multimedia and Process Control: Some Initial Experimental Results', *Computers and Graphics* , 17(3) , pp. 205-218.

Alty, J.L.,Bergan, M. 1995. Multi-media Interfaces for Process Control: Matching Media to Tasks" , *Control Engineering Practice* , 3(2) , pp. 241-248.

America, P., 1987. POOL-T: A parallel object-oriented language. *Object-Oriented Concurrent Programming* Tokoro, M and Yonezawa, A (Ed.) MIT Press.

Anderson, K.M., Taylor, R.M., Whitehead, E.J., 1994, September. Chimera: Hypertext for Heterogeneous Software Environments. *Proc. ACM Hypertext '94*. Edinburgh. Pp. 94-107.

Anderson, T., 1998. Beyond Eisenstein. A case study in Interactive Television. *Interactive Multimedia. Visions of Multimedia for Developerrs, Educators and Information Providers*. Microsoft Press, Redmond. pp. 193-213

Andrews, G.R., 1991.*Concurrent Programming Principles and Practice*. Benjamin Cummins. pp. 494-509.

Arons, B., 1991. December. Hyperspeech: Navigating in Speech-Only Hypermedia. In *Proceedings: ACM Hypertext '91*, San Antonio, TX, pp. 133 - 146.

Bal, H.E., Grune, R., 1994. *Programming Language Essentials*, Addison Wesley.

Bell, D., Parr. M., 1998. *Java for Students*. ISBN 0-13-858440-0, Prentice Hall Europe.London.

Benguelin, A. Dongarra, J.,Geist, A., Marrichek, R.,Sunderam, V., 1990. *User's Guide to PVM Parallel Virtual Machine*, Oak Ridge National Laboratory Report ORNL/TM-11826.

Benyon, D., 1992. The Role of Task Analysis in Systems Design, *Interacting with Computers 4,* pp. 102-123.

Bergman, R.E., Moore, T.V., 1990. *Managing interactive video/multimedia projects*. Educational Technology Publications.

Berners-Lee, T. Calliau, R. Groff, J.,Pollerman, P. 1992. *World-Wide Web: The Information Universe*. CERN.

Berners-Lee, T.,Conolly, D., 1995. Hypertext Mark-up Language Specification 2.0. *RFC1866*Available at http://sunsite.doc.ic.ac.uk/rfc/rfc1866.txt.

Bertrand, F., Colaitis, F. Leger A.., 1992. The MHEG Standard and its Relation with the Multimedia and Hypermedia Area. *Proc. IEE Conference on Image Processing and its Application*.

Bhushan. A.K. 1972., File Transfer Protocol (FTP): *RFC 0414*. Available at http://sunsite.doc.ic.ac.uk/rfc/rfc0414.txt.

Bieber, P., 1996 April. Interpretation d'un modele de securite. *Techniques et Sciences Informatiques*, 15(6), Editions Hermes.

Bieber, P., 1996, December. Formal Techniques for an ITSEC-E4 Secure Gateway. *Proc. 12th Annual Computer Security Applications Conference*, IEEE computer society press.

Boll, S., Lohr, M., 1996, March. Interactive Multimedia Presentation Capabilities for an Object-Oriented DBMS, *9th ERCIM Database Research Group Workshop on Multimedia Database Systems* Darmstadt, Germany.

Bornat, R., Sufrin, B.A., 1994, August. The Gist of Jape. Oxford University Programming Research Group Research Monograph.

Brachman, R., Anand, D., 1994, August. The process of knowledge discovery in databases: A first sketch. *Proc AAAI KDD-94 Workshop,* Seattle. pp. 1-11.

Burns, A., Welling, A., 1990. *Real-Time Systems and their Programming Languages.* Addison-Wesley, London.

Burrill, V.A., Kirste, T., Weiss, J.M., 1994, April. Time-varying sensitive regions in dynamic multimedia objects: a pragmatic approach to content-based retrieval from video, *Information and Software Technology Journal* 36(4), Butterworth-Heinemann ,pp. 213-224.

Cambell, B.,Goodman. J.M., 1988, July. HAM: A general-purpose hypertext abstract machine. *Comm. ACM,* 31(7), pp856-861.

Caromel, D., 1990. Programming Abstractions for Concurrent Programming. Proc. *TOOLS Pacific 90.* Sydney.

Caromel, D., 1993, September. Towards a method of Object-Oriented Concurrent Programming. *Communications of the ACM 36,*9, pp 90-101.

Carriero, N. Gerlemter, D., 1989, April. Linda in Context. *Communications of the ACM 32.*

Casner, S.M., 1991. A Task-Analytic Approach to the Automated Design of Graphic Presentations. *ACM Transactions on Graphics 10,* pp.111-151.

Chang, C. ,Sussman, A., Salz, J., 1995, January. *Support for Distributed Dynamic Data Structures in C++,* University of Maryland: Department of Computer Science Technical Report CS-TR-3416 and UMIACS-TR-95-19.

Chiueh.T.C., 1994. Content-based image indexing. *Proc. VLDB 94 Conference.* pp. 582-593.

Conklin. J., 1987, September. Hypertext: An Introduction and Survey. *IEEE Computer.* pp 17-41.

Cook, S. Kohoutkova, J., Jeffrey, K., 1996, March. Hypermedata: Meta-structures for Exchanging Hypermedia Documents. , *9th ERCIM Database Research Group Workshop on Multimedia Database Systems* Darmstadt, Germany.

Copas, C.V. and Edmonds, E.A., 1994. Executable Task Analysis: Integration Issues', *People and Computers IX* , Cockton, G., Draper, S.W., and Weir, G.R.S. (eds), Cambridge University Press. , pp. 339-352

Cypher, A. Stelzner, M., 1991. Graphical knowledge-based model editors. In: J. W Sullivan & S. W Tyler (eds): *Intelligent user interfaces*, New York/Reading MA: ACM Press, Addison Wesley. pp. 403-420.

Department of Defense, 1994. *Continuous Acquisition and Life-Cycle Support (CALS) Implementation Guide.* MIL-HDBK-59B.

Department of Defense, 1993. *Contractor Integrated Technical Information Service (CITIS).* MIL-STD-974.

Dijkstra, E.W. 1976., *A Discipline of Programming.* Prentice-Hall.

Dijkstra, E.W., 1968, March. Goto Statement Considered Harmful, *Comm. ACM.*

Dimitrova, N. Golshani, F., 1994. RX for Semantics Video Database Retrieval, *Proc. ACM Multimedia '94.*

Edwards, D.M., Hardman, L., 1989. 'Lost in Hyperspace': Cognitive mapping and navigation in a hypertext environment. E. McAleese (ed) *Hypertext: theory into practice.* Intellect. Oxford..

van Eijk P.H.J., Vissers, C.A.,Diaz M.(eds), 1989.*The formal description technique Lotos*, North Holland.

Eun, S., No, E.S., Kim H.C.,Yoon, H., Maeng S.R., 1994. Eventor: An Authoring System for Interactive Multimedia Applications. *Multimedia Systems* 2. pp. 129-140.

Fallenstein-Hellman, M.F., James, W.R., 1995 *The Multimedia Casebook.* VNR.

Faloutsos, C., Barber, R., Flickner, M.,Hafner, J., Niblack, W.,Petkovic, D., 1994 Efficient and effective querying by image content. *Journal of Intelligent Information Systems. 3.* pp. 231-262.

Faraday.,P. Sutcliffe. A., 1993. A Method for Multimedia Interface Design , ACM Computer Graphics, *People and Computers VIII.* pp. 173-190.

Farrington, G., 1994. Air maintenance task oriented support system (AMTOSS). *Proc. Communicating '94* .

Fayyad, U., Uthurusamy, R.(eds.), 1995, August. *Proc. 1st Intenational Conference on Knowledge Discovery and Data Mining,* Menlo Park, Calif.

Fencott, C., 1996. *Formal Methods for Concurrency,* ISBN 1-85032-173-6, International Thompson Computer Press, London. pp.281-282.

Fischer, D., 1997. *A theory of presentation and its implications for the design of online technical documentation.* Ph.D. Thesis. Coventry University.

Fischer, D., Richards, C. J. , 1995.The presentation of time in interactive animated systems diagrams In: Rae A Earnshaw & John A Vince (eds): *Multimedia systems & applications.* London/San Diego: Academic Press.

Fischer, D., Heino, I., Cotterel, D., 1996. *OMIMO System: User Requirements.* Project Deliverable D3, OMIMO Project, Telematics Applications Programme, IE2054

Fountain, A., Hall, W., Heath, I., Davis, H. C., 1990, November. Microcosm: An Open Model for Hypermedia With Dynamic Linking. In Rizk, A., Streitz, N., Andre, J. (eds.), *Hypertext: Concepts, systems and Applications, Proceedings of the Hypertext '90 Conference,* INRIA, France. pp. 298-311.

Fox. E., 1991, October. Advances in Interactive Digital Multimedia Systems, *IEEE Computer.*

Gehani, N. H., Roome, W. D., 1988, December. Concurrent C++: Concurrent programming with class(es). *Software - Practice and Experience* 16,12.

Geissler. J., 1996, March. Navigating Hypermovies. *9th ERCIM Database Research Group Workshop on Multimedia Database Systems* Darmstadt, Germany.

Goose. S., 1997, July. *A Framework for Distributed Open Hypermedia.* PhD. Thesis, University of Southhampton.

Gronbaek, K. Trigg. R. H., 1992, November. Design Issues for a Dexter-Based Hypermedia System. *Proc ACM Hypertext '92 Conference,* Milano, Italy. pp. 191-200.

Gu, J., Heuhold, E.J., 1993, November. A data model for multimedia information retrieval. *Proc. First International Conference on Multi-Media Modeling,* World Scientific, Singapore. pp.113-127.

Gupta, A., Weymouth, T., Jian. R., 1991, September. Semantic queries with pictures: The VIMSYS model. *Proc International Conference on Very Large Databases '91,* Barcelona. pp. 69-79.

Haan B.J., Kahn, P.,Riley, V.A., Coombs, J.H, Meyrowitz, N., 1992, January. IRIS Hypermedia Services. *Comm. ACM,* 35(1). pp. 36-51.

Halasz, F.G., 1988, July. Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems, *Comm. ACM,* 31(7). pp. 836-852.

Halasz, F.G., Schwartz, M., 1990. The Dexter Hypertext Reference Model. *Proceedings of the NIST Hypertext Standardisation Workshop, NIST Special Publication SP500-178,* pp. 95-133.

Halasz, F.G., Schwartz. M., 1994. The Dexter Hypertext Reference model, *Comm. ACM*, 31(2). pp. 30-39.

Hardman, L.,van Rossum, G., Bulterman, D.C.A. 1993, August. Structured Multimedia Authoring. *Proc.ACM Multimedia 93*, pp. 283-290.

Hardman, L. Bulterman, D.C.A., 1995, November. Using the Amsterdam Hypermedia Model for Abstracting Presentation Behavior, *ACM Workshop on Effective Abstractions in Multimedia,* San Francisco.

Hinson, D. 1991. Computer-aided fault-finding documentation. *Communicator,* Vol. 2, No. 3, pp. 2-9.

Hoare, C. A. R., 1969, October. An axiomatic basis for computer programming. *Communications of the ACM 12*. pp. 576-583.

Hoare, C. A. R., 1978. Communicating Sequential Processes. *Comm. ACM* 21(8). pp. 666-677.

Hoare, C. A. R., 1972. Towards a theory of parallel programming, In *Operating Systems Techniques*, C.A.R Hoare, RH.Perrot (eds), Academic Press.

Hogg , R. 1995. *Teaching notes*

http://www.sunderland.ac.uk/~ts0bho/lect_mat/mm_lev_1/flowcht.htm.

Horton, W. K., 1993. Let's do away with manuals.... *Communicator,* Vol. 4, No. 4, pp. 18-21.

Horton, W. K., 1990. *Designing & writing online documentation: help files to hypertext.* John Wiley & Sons.

Hwang, Y-S., Moon. B., Sharma, S.D., Ponnusang, R., Das, R., Salz, J.H., 1995 June. Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines. *Software practice and Experience* 25, 6.

IETM 1992., *Data Base, Revisable: Interactive Electronic Technical Manuals, For The Support Of.* MIL-D-87269.

INMOS Ltd., 1984. *Occam Programming Manual.* Prentice-Hall Int., Englewood Cliffs, NJ.

International Standards Organisation, 1992. *Hypermedia/Time-based Structuring Language (HyTime),* ISO/IEC Standard 10744.

Jeffcoate. J., 1995. *Multimedia in Practice, Technology and Applications.* Prentice-Hall International.

Johnson, P., Johnson, H., Waddington, R., Shouls, A., 1988. Task Related Knowledge Structures: Analysis, Modelling and Application. *People and Computers IV,* Cambridge University Press.

Johnson, P, Johnson, H., 1991. Knowledge Analysis of Tasks: task analysis and specification for human-computer systems. A. Downton (ed), *Engineering the Human Computer Interface*, McGraw Hill, London.

Jones, C.B., 1990. *Systematic Software Development Using VDM (2nd Edition),* Prentice-Hall.

Jones, S., 1991. *Text and Context. Document processing and storage.* London Springer-Verlag.

Kacmar, C. J., Leggett, J. J., 1991. A Process-Oriented Extensible Hypertext Architecture. *ACM Transactions on Information Systems*, 9(4). pp 399-419.

Kafura, D., Lee, K. H., 1990. ACT++: Building a concurrent C++ with Actors, *Journal of Object Oriented Programming, 3, 1.*

Knappe, F., Pani, G., Schnable, F., 1993. The Architecture of a Massively Distributed Hypermedia System.*Internet Research: Electronic Networking Applications and Policy,* 3(1). pp. 10-24.

Kuntz. M., 1996, March. Mining Multimedia Data: New Problems and Interaction based solutions. *9th ERCIM Database Research Group Workshop on Multimedia Database Systems* Darmstadt, Germany.

Leggett, J. J., Schnase, J. L.., 1991. Dexter with Open Eyes. *Comm ACM,* 37(2). pp. 77-86.

Lin, C. K-I., 1995. FastMap: A fast algorithm for indexing, data-mining and visualisation of traditional and multimedia datasets. *Proc. SIGMOD95 Conference*. pp. 163-174.

Matsuoka, S. Wakita, K. Yonezawa, A., 1993. Inheritance Anomaly in object-oriented concurrent programming languages. *Research Directions in Object-Based Concurrency,* G. Agha, P. Wegner, A Yonezawa Eds, MIT Press.

Mayfield, J., Nicholas, C., 1993. SNITCH: Augmenting hypertext documents with a semantic net. *International Journal of Intelligent and Co-operative Information Systems,* 2(3). pp. 335-351.

McCarthy, J., 1960, April. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Comm. ACM*, 3(4). pp. 185-195.

Mehrotra, R., Gary, J., 1995, March. Feature-index-based similar shape retrieval. *Proc. IFIP 2.6 Conference of Visual Database Systems,* Lausanne.

Meyer, B., 1993, September. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM 36(9)*

Mil-M-87268 (GCSFUI), 1992 *Interactive electronic technical manuals. General content, style, format and user-interaction requirements.*

Milner, R., 1989. *Communication and Concurrency.* Prentice-Hall, Hemel Hempstead.

Moller, F., 1992. *The Edinburgh concurrency workbench (version 6.1).Technical Report, User Manual.*Laboratory for the Foundations of Computer Science, University of Edinburgh.

Myers, J., 1994. Post Office protocol (POP3):*RFC1734.* Available at http://sunsite.doc.ic.ac.uk/rfc/rfc1734.txt.

Newman, R., 1990, June. Learner Workstation Hardware Specification, in *Definition of Standard Training Tools for Industrial Training Environments and an Outline Functional Specification for a Learner Workstation*, DELTA Project D1004/P7064.

Newman, R., 1993, March. A crystallisation model for program visualisation, *Proc Visual Aspects of Man-Machine Systems, 93* Ed. J Polak, Prague.

Newman, R., Richards, C. J., Fischer, D., Patera, V. Heino, I. Virta, H. Koelling, U., Cotterell, M., 1997, May. *OMIMO Final Report.* Telematics Applications Programme, Information Engineering sector, Project IE2054.

Newman, R., Payne, R., 1994, September. Integration of Object Oriented and Concurrent Programming. *Proc Euromicro 94*, Liverpool.

Newman, R., Payne, L., Monk, K., 1995, August. Teaching Transferable Programming Skills for Varied Subject Areas, *3rd Annual Conference on Teaching of Computing*, Dublin.

Newman. R., 1995, September. Synchronised Method Calls and Verification of Concurrent Object Oriented Programs, *Proc Euromicro 95*, Como.

Newman, R., 1996, September. Using Java for Real Time Systems, *Short Papers Proceedings, Euromicro 96* Prague.

Newman, R., 1996, December. From Engineering Data to Documentation: Managing the Authoring Process, *Applications for the European Information Society, Telematics Applications Programme Conference,* Brussels.

Newman, R., 1998. The ClassiC Programming Language and Design of Synchronous Concurrent Object Oriented Languages, *Journal of Systems Architecture*, Elsevier Scientific, In Press.

Newman, R., 1998, September, A methodology for design of large hypermedia systems, *Euromicro '98*, Västerås, Sweden (accepted for presentation).

Newman, R., Gatward, R., Poppleton, M., 1994. Paradigms for Teaching Introductory Programming, Proc. Software Engineering in Higher Education, Southampton.

Nielson, J., 1990, March. The Art of Navigating through Hypertext, *Comm. ACM,* 33(3). pp. 296-310.

Nye, A., O'Reilly, T., 1990. *X Toolkit Intrinsics Programming Manual.* O'Reilly and Associates, Inc, Sebastopol, CA.

Orlowski, 1995. BISAM. *Broadband integrated services for aircraft maintenance.* Lufthansa internal presentation, FRA OB/M 24/4/95.

Pearl, A., 1991, November. Sun's Link Service: A Protocol for open Linking. *Proc. Hypertext '89,* Pittsburgh, PA.. pp 137-146.

Petrakis, E.G., Orphanoudakis, S.C., 1993, October. Methodology for the Representation, Indexing and Retrieval of Images by Content, *Image and Vision Computing,* 11(8). pp. 504-521.

Pruckler, T.,Schreff, M., 1996, March. An Architecture of a Hypermedia DBMS Supporting Physical Data Independence, *9th ERCIM Database*

Research Group Workshop on Multimedia Database Systems Darmstadt, Germany.

Rizk, A., Sauter, I., 1992, November. Multicard: An Open Hypermedia Syatem. *Proc ACM Hypertext '92 Conference,* Milano, pp. 4-10.

van Rossum, G., Jansen, J. Mullender, K.S. , Bulterman, D.C.A., 1993, August. CMIFed: a presentation environment for portable hypermedia documents. *Proc ACM Multimedia '93*, Anaheim CA.. pp. 183 – 188.

Rubens, P., Krull, R., 1988. Designing Online Information. Barrett E (ed.). *Text, ConText, and Hypertext.* The MIT Press, Cambridge, MA.. pp. 291-309.

Schnase, J. L., Leggett, J. J., Hicks, D. L., Szabo, R. L., 1993. Semantic data modelling of hypermedia associations. *ACM Transactions on Information Systems,* 11(1). pp. 27-50.

Schwier, R. A., Misanchuk, E. R., 1993. *Interactive Multimedia Instruction.* Educational Technology Publications, Englewood Cliffs. ISBN 0-87778-251-2. pp. 294-296.

SITA., 1996, August. Aeronet – propelling the industry forward. *SITA Global Network News*, 8/96. pp 1-8.

Spivey, J. M., 1989.*The Z Notation- A Reference Manual*, Prentice-Hall,London. ISBN 0-13-983768X

Stirling, C., 1991. An introduction to modal and temporal logics for CCS. *Lecture notes in Computer Science*, (491) pp. 2-20.

Sun Microsystems, Inc., 1995. *The Java Language: A White Paper.* http://java.sun.com/1.0alpha3/doc/overview/java/index.html.

Sun Microsystems, Inc., 1996. The Java Language Specification. http://java.sun.com/newdocs.html#dev.

Sutcliffe, A., Faraday, P., 1994. Systematic design for task-related multimedia interfaces. *Information and Software Technology* 36(4).

Taylor, J. C., 1990. Organizational context for aircraft maintenance and inspection. *Proceedings of the Human Factors Society 34th Annual Meeting*, Vol. 2. pp. 1176-1180.

Ueda, H., 1994. Automatic Structure Visualisation for Video Editing. *Proc. INTERCHI '93.* pp. 137-141.

Ventura C., 1988. Why Switch from Paper to Electronic Manuals? *Proc. ACM Conference on Document Processing Systems*, Santa Fe, New Mexico. pp. 111-116.

VTT Automation, Electronics and Information Technology, 1996, November. *Multimedia based maintenance support.*

Wiil, U. K., Leggett, J. J., 1996, March. The HyperDisco Approach to Open Hypermedia Systems. *Proc. ACM Hypertext '96 Conference,* Washington D.C., U.S.A., pp 140-148.

Wu, G., Baird, S., Robinson, B., 1997. *HyTech - A HyTime Application*, EDRC Research Report, University of Hertfordshire.

Wu, J. K., Narasihalu, A. D., Mehtre, B. M., Lam, C. P, Gao, Y. J., 1995, February.CORE: A Content-based Retrieval Engine for Multimedia Information Systems, *Multimedia Systems* 3(1). pp. 25-41.

de Young, L., 1990. 'Linking considered harmful'. In: N Streitz, A Rizk J Andrei (eds): *Hypertext: Concepts, systems and applications*, Cambridge University Press. pp. 238-249.

Yu, J., Xiang, Y., 1995. Hypermedia Presentation and Authoring System, *Hyper Proceedings, 6th International WorldWide Web Conference,* at http://www6.nttlabs.com/HyperNews/get/PAPER91.html